

This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images,
please do not report the images to the
Image Problem Mailbox.**

THIS PAGE BLANK (USPTO)

1
2
3
4
5
6
7
8
9
10

Internal Working Document
Released for informational purposes only

11 **Common Language Infrastructure (CLI)**
12 **Partition I:**
13 **Concepts and Architecture**

14
15
16
17
18
19
20
21
22
23

24 The following have participated in the work of ECMA TC39/TG3 and their contributions are gratefully acknowledged:
25 Fujitsu Software Corporation, Hewlett-Packard, Intel Corporation, IBM Corporation, ISE, Microsoft Corporation, Monash University, Netscape,
26 Phone.Com, Plum Hall, Sun Microsystems

27

1

Table of Contents

2	1	Scope	6
3	2	Conformance	7
4	3	References	8
5	4	Glossary	9
6	5	Overview of the Common Language Infrastructure	24
7	5.1	Relationship to Type Safety	24
8	5.2	Relationship to Managed Metadata-driven Execution	25
9	5.2.1	Managed Code	25
10	5.2.2	Managed Data	26
11	5.2.3	Summary	26
12	6	Common Language Specification (CLS)	27
13	6.1	Introduction	27
14	6.2	Views of CLS Compliance	27
15	6.2.1	CLS Framework	27
16	6.2.2	CLS Consumer	27
17	6.2.3	CLS Extender	28
18	6.3	CLS Compliance	28
19	6.3.1	Marking Items as CLS-Compliant	29
20	7	Common Type System	30
21	7.1	Relationship to Object-Oriented Programming	32
22	7.2	Values and Types	32
23	7.2.1	Value Types and Reference Types	32
24	7.2.2	Built-in Types	32
25	7.2.3	Classes, Interfaces and Objects	33
26	7.2.4	Boxing and Unboxing of Values	34
27	7.2.5	Identity and Equality of Values	34
28	7.3	Locations	35
29	7.3.1	Assignment Compatible Locations	35
30	7.3.2	Coercion	35
31	7.3.3	Casting	36
32	7.4	Type Members	36

1	7.4.1	Fields, Array Elements, and Values	36
2	7.4.2	Methods	36
3	7.4.3	Static Fields and Static Methods	37
4	7.4.4	Virtual Methods	37
5	7.5	Naming	37
6	7.5.1	Valid Names	37
7	7.5.2	Assemblies and Scoping	38
8	7.5.3	Visibility, Accessibility, and Security	39
9	7.6	Contracts	41
10	7.6.1	Signatures	42
11	7.7	Assignment Compatibility	45
12	7.8	Type Safety and Verification	45
13	7.9	Type Definers	45
14	7.9.1	Array Types	46
15	7.9.2	Unmanaged Pointer Types	48
16	7.9.3	Delegates	48
17	7.9.4	Interface Type Definition	48
18	7.9.5	Class Type Definition	49
19	7.9.6	Object Type Definitions	51
20	7.9.7	Value Type Definition	53
21	7.9.8	Type Inheritance	54
22	7.9.9	Object Type Inheritance	54
23	7.9.10	Value Type Inheritance	54
24	7.9.11	Interface Type Inheritance	54
25	7.10	Member Inheritance	55
26	7.10.1	Field Inheritance	55
27	7.10.2	Method Inheritance	55
28	7.10.3	Property and Event Inheritance	55
29	7.10.4	Hiding, Overriding, and Layout	55
30	7.11	Member Definitions	56
31	7.11.1	Method Definitions	57
32	7.11.2	Field Definitions	57
33	7.11.3	Property Definitions	57
34	7.11.4	Event Definitions	58
35	7.11.5	Nested Type Definitions	59
36	8	CLI Metadata	60
37	8.1	Components and Assemblies	60

1	8.2	Accessing Metadata	60
2	8.2.1	Metadata Tokens	60
3	8.2.2	Member Signatures in Metadata	61
4	8.3	Unmanaged Code	61
5	8.4	Method Implementation Metadata	61
6	8.5	Class Layout	61
7	8.6	Assemblies: Name Scopes for Types	62
8	8.7	Metadata Extensibility	63
9	8.8	Globals, Imports, and Exports	64
10	8.9	Scoped Statics	64
11	9	Name and Type Rules for the Common Language Specification	65
12	9.1	Identifiers	65
13	9.2	Overloading	65
14	9.3	Operator Overloading	66
15	9.3.1	Unary Operators	66
16	9.3.2	Binary Operators	67
17	9.3.3	Conversion Operators	68
18	9.4	Naming Patterns	68
19	9.5	Exceptions	69
20	9.6	Custom Attributes	69
21	10	Collected CLS Rules	71
22	11	Virtual Execution System	74
23	11.1	Supported Data Types	74
24	11.1.1	Native Size; native int, native unsigned int, O and &	75
25	11.1.2	Handling of Short Integer Data Types	76
26	11.1.3	Handling of Floating Point Datatypes	76
27	11.1.4	CIL Instructions and Numeric Types	79
28	11.1.5	CIL Instructions and Pointer Types	80
29	11.1.6	Aggregate Data	81
30	11.2	Module Information	84
31	11.3	Machine State	84
32	11.3.1	The Global State	84
33	11.3.2	Method State	85
34	11.4	Control Flow	88
35	11.4.1	Method Calls	89
36	11.4.2	Exception Handling	92

1	11.5	Proxies and Remoting	97
2	11.6	Memory Model and Optimizations	97
3	11.6.1	The Memory Store	97
4	11.6.2	Alignment	97
5	11.6.3	Byte Ordering	98
6	11.6.4	Optimization	98
7	11.6.5	Locks and Threads	98
8	11.6.6	Atomic Reads and Writes	99
9	11.6.7	Volatile Reads and Writes	99
10	11.6.8	Other Memory Model Issues	99
11	11.7	Atomicity of Memory Accesses	100
12	12	Index	101

1 1 **Scope**

2 This ECMA Standard defines the Common Language Infrastructure (CLI) in which applications written in
3 multiple high level languages may be executed in different system environments without the need to rewrite the
4 application to take into consideration the unique characteristics of those environments. This ECMA Standard
5 consists of several sections in order to facilitate understanding various components by describing those
6 components in their separate sections. These sections are:

7 Partition I: Architecture

8 Partition II: Metadata Definition and Semantics

9 Partition III: CIL Instruction Set

10 Partition IV: Profiles and Libraries

11 Partition V: Annexes

1 2 **Conformance**

2 A system claiming conformance to this ECMA Standard shall implement all the mandatory requirements of
3 this standard, and shall specify the profile (see Partition IV) that it implements. The minimal implementation is
4 the Kernel Profile (see Partition JV). A conforming implementation may also include additional functionality
5 that does not prevent running code written to rely solely on the profile as specified in this standard. For
6 example, it may provide additional classes, new methods on existing classes, or a new interface on a
7 standardized class, but it shall not add methods or properties to interfaces specified in this standard.

8 A compiler that generates Common Intermediate Language (CIL, see Partition III) and claims conformance to
9 this ECMA Standard shall produce output files in the format specified in this standard and the CIL it generates
10 shall be valid CIL as specified in this standard. Such a compiler may also claim that it generates *verifiable*
11 code, in which case the CIL it generates shall be verifiable as specified in this standard.

1 3 **References**

2 IEC 60559:1989, Binary Floating-point Arithmetic for Microprocessor Systems (previously designated IEC
3 559:1989)

4 ISO/IEC 10646 (all parts), Information technology — Universal Multiple-Octet Coded Character Set (UCS).

5 The Unicode Consortium. The Unicode Standard, Version 3.0, defined by: The Unicode Standard, Version 3.0
6 (Reading, MA, Addison-Wesley, 2000. ISBN 0-201-61633-5), and Unicode Technical Report #15: Unicode
7 Normalization Forms.

8 ISO/IEC 646:1991 Information technology -- ISO 7-bit coded character set for information interchange

9 ISO/IEC 11578:1996 (E) Information technology - Open Systems Interconnection - Remote Procedure Call
10 (RPC), Annex A: Universal Unique Identifier

11 Federal Information Processing Standard (FIPS 180-1), Secure Hash Standard (SHA-1), 1995 April 7.

12 Extensible Markup Language (XML) 1.0 (Second Edition), 2000 October 6, <http://www.w3.org/TR/2000/REC-xml-20001006>

14 Network Working Group. RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1. R. Fielding, J. Gettys, J.
15 Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. 1999 June, <ftp://ftp.isi.edu/in-notes/rfc2616.txt>

16 Network Working Group. RFC 2617: HTTP Authentication: Basic and Digest Access Authentication. J.
17 Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, L. Stewart. 1999 June,
18 <ftp://ftp.isi.edu/in-notes/rfc2617.txt>

19 IETF (Internet Engineering Task Force). RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. T.
20 Berners-Lee, R. Fielding, L. Masinter. 1998 August, <http://www.ietf.org/rfc/rfc2396.txt>.

21 Network Working Group. RFC-1222: Advancing the NSFNET Routing Architecture. H-W Braun, Y. Rekhter.
22 1991 May, <ftp://ftp.isi.edu/in-notes/rfc1222.txt>

1 4 Glossary

2 For the purpose of this ECMA Standard, the following definitions apply. They are collected here for ease of
3 reference, but the definition is presented in context elsewhere in the specification, as noted. Definitions
4 enclosed in square brackets [] were not extracted from the body of the standard.

5 The remainder of this section and its subsections contain only informative text

Term	Description	Pt	Ch	Section
Abstract	Only an abstract object type is allowed to define method contracts for which the type or the VES does not also provide the implementation. Such method contracts are called abstract methods	I	<u>7.9.6.2</u>	Concreteness
Accessibility of members	A type scopes all of its members, and it also specifies the accessibility rules for its members. Except where noted, accessibility is decided based only on the statically visible type of the member being referenced and the type and assembly that is making the reference. The CTS supports seven different rules for accessibility: Compiler-Controlled; Private; Family; Assembly; Family-and-Assembly; Family-or-Assembly; Public.	I	<u>7.5.3.2</u>	Accessibility of Members
Aggregate data	Data items that have sub-components (arrays, structures, or object instances) but are passed by copying the value. The sub-components can include references to managed memory. Aggregate data is represented using a <i>value type</i> ...	I	<u>11.1.6</u>	Aggregate Data
Application domain	A mechanism ... to isolate applications running in the same operating system process from one another.	I	<u>11.5</u>	Proxies and Remoting
Array elements	The representation of a value (except for those of built-in types) can be subdivided into sub-values. These sub-values are either named, in which case they are called fields , or they are accessed by an indexing expression, in which case they are called array elements .	I	<u>7.4.1</u>	Fields, Array Elements, and Values
Argument	[Value of an operand to a method call]			
Array types	Types that describe values composed of array elements are array types .	I	<u>7.4.1</u>	Fields, Array Elements, and Values
Assembly	An assembly is a configured set of loadable code modules and other resources that together implement a unit of functionality.	I	<u>7.5.2</u>	Assemblies and Scoping
Assembly scope	Type names are scoped by the assembly that contains the implementation of the type.... The type name is said to be in the assembly scope of the assembly that implements the type.	I	<u>7.5.2</u>	Assemblies and Scoping
Assignment compatibility	Assignment compatibility of a value (described by a type signature) to a location (described by a location signature) is defined as follows: One of the types supported by the exact type of the value is the same as the type in the location signature.	I	<u>7.7</u>	Assignment Compatibility
Attributes	<i>Attributes</i> of types and their members attach descriptive information to their definition	II	<u>5.9</u>	Attributes and Metadata

	information to their definition.			Metadata
Base Class Library	This Library is part of the Kernel Profile. It is a simple runtime library for a modern programming language.	IV	<u>5.1</u>	Runtime Infrastructure Library
Binary operators	Binary operators take two arguments, perform some operation and return a value. They are represented as static methods on the class that defines the type of one of their two operands or the return type.	I	<u>9.3.2</u>	Binary Operators
Boolean Data Type	A CLI Boolean type occupies one byte in memory. A bit pattern of all zeroes denotes a value of false. A bit pattern with any bit set (analogous to a non-zero integer) denotes a value of true.	III	<u>1.1.2</u>	Boolean Data Type
Box	The box instruction is a widening (always typesafe) operation that converts a value type instance to System.Object by making a copy of the instance and embedding it in a newly allocated object.	I	<u>11.1.6.2.5</u>	Boxing and Unboxing
Boxed type	For every Value Type, the CTS defines a corresponding Reference Type called the boxed type .	I	<u>7.2.4</u>	Boxing and Unboxing of Values
Boxed value	The representation of a value of a boxed type (a boxed value) is a location where a value of the Value Type may be stored.	I	<u>7.2.4</u>	Boxing and Unboxing of Values
Built-in types	..Data types [that] are an integral part of the CTS and are supported directly by the Virtual Execution System (VES).	I	<u>7.2.2</u>	Built-In Types
By-ref parameters	The address of the data is passed from the caller to the callee, and the type of the parameter is therefore a managed or unmanaged pointer.	I	<u>11.4.1.5</u>	Parameter Passing
By-value parameters	The value of an object is passed from the caller to the callee	I	<u>11.4.1.5</u>	Parameter Passing
Calling Convention	A calling convention specifies how a method expects its arguments to be passed from the caller to the called method.	II	<u>14.3</u>	Calling Convention
Casting	Since a value can be of more than one type, a use of the value needs to clearly identify which of its types is being used. Since values are read from locations that are typed, the type of the value which is used is the type of the location from which the value was read. If a different type is to be used, the value is cast to one of its other types.	I	<u>7.3.3</u>	Casting
CIL	[Common Intermediate Language]			
Class contract	A class contract specifies the representation of the values of the class type. Additionally, a class contract specifies the other contracts that the class type supports, e.g., which interfaces, methods, properties and events shall be implemented.	I	<u>7.6</u>	Contracts
Class type	A complete specification of the representation of the values of the class type and all of the contracts (class, interface, method, property, and event) that are	I	<u>7.9.5</u>	Class Type Definition

	interface, method, property, and event) that are supported by the class type.			
CLI	At the center of the Common Language Infrastructure (CLI) is a single type system, the Common Type System (CTS), that is shared by compilers, tools, and the CLI itself. It is the model that defines the rules the CLI follows when declaring, using, and managing types.	I	<u>5</u>	Overview of the Common Language Infrastructure
CLS	The Common Language Specification (CLS) is a set of conventions intended to promote language interoperability.	I	<u>6</u>	Common Language Specification (CLS)
CLS (consumer)	A CLS consumer is a language or tool that is designed to allow access to all of the features supplied by CLS-compliant frameworks (libraries), but not necessarily be able to produce them.	I	<u>6</u>	Common Language Specification (CLS)
CLS (extender)	A CLS extender is a language or tool that is designed to allow programmers to both use and extend CLS-compliant frameworks.	I	<u>6</u>	Common Language Specification (CLS)
CLS (framework)	A library consisting of CLS-compliant code is herein referred to as a "framework".	I	<u>6</u>	Common Language Specification (CLS)
Code labels	Code labels are followed by a colon (":") and represent the address of an instruction to be executed	II	<u>5.4</u>	Labels and Lists of Labels
Coercion	Coercion takes a value of a particular type and a desired type and attempts to create a value of the desired type that has equivalent meaning to the original value.	I	<u>7.3.2</u>	Coercion
Common Language Specification (CLS)	The Common Language Specification (CLS) is a set of conventions intended to promote language interoperability.	I	<u>6</u>	Common Language Specification (CLS)
Common Type System (CTS)	[I] The Common Type System (CTS) provides a rich type system that supports the types and operations found in many programming languages.	I	<u>5</u>	Overview of the Common Language Infrastructure
Compiler-controlled accessibility	Accessible only through use of a definition, not a reference, hence only accessible from within a single compilation unit and under the control of the compiler.	I	<u>7.5.3.2</u>	Accessibility of Members
Compound types	Types that describe values composed of fields are compound types.	I	<u>7.4.1</u>	Fields, Array Elements, and Values
Computed destinations	The destination of a method call may be either encoded directly in the CIL instruction stream (the <code>call</code> and <code>jmp</code> instructions) or computed (the <code>callvirt</code> , and <code>calli</code> instructions).	I	<u>11.4.1.3</u>	Computed Destinations
Concrete	An object type that is not marked <code>abstract</code> is by definition <code>concrete</code> .	I	<u>7.9.6.2</u>	Concreteness
Conformance	A system claiming conformance to this ECMA Standard shall implement all the mandatory requirements of this Standard.	I	<u>2</u>	Conformance

e	shall implement all the mandatory requirements of this standard, and shall specify the profile that it implements.			
Contracts	Contracts are named. They are the shared assumptions on a set of signatures ... between all implementers and all users of the contract.	I	<u>7.6</u>	Contracts
Conversion operators	Conversion operators are unary operations that allow conversion from one type to another. The operator method shall be defined as a static method on either the operand or return type.	I	<u>9.3.3</u>	Conversion Operators
Custom Attributes	Custom attributes add user-defined annotations to the metadata. Custom attributes allow an instance of a type to be stored with any element of the metadata.	II	<u>20</u>	Custom Attributes
Custom modifiers	Custom modifiers, defined using modreq ("required modifier") and modopt ("optional modifier"), are similar to custom attributes ...except that modifiers are part of a signature rather than attached to a declaration. Each modifier associates a type reference with an item in the signature.	II	<u>7.1.1</u>	modreq and modopt
Data labels	Data labels specify the location of a piece of data	II	<u>5.4</u>	Labels and Lists of Labels
Delegates	Delegates are the object-oriented equivalent of function pointers. Delegates are created by defining a class that derives from the base type System.Delegate	I	<u>7.9.3</u>	Delegates
Derived Type	A derived type guarantees support for all of the type contracts of its base type. A type derives directly from its specified base type(s), and indirectly from their base type(s).	I	<u>7.9.8</u>	Type Inheritance
Enums	An <i>enum</i> , short for <i>enumeration</i> , defines a set of symbols that all have the same type.	II	<u>13.3</u>	Enums
Equality	For value types, the equality operator is part of the definition of the exact type. Definitions of equality should obey the following rules: <ul style="list-style-type: none"> • Equality should be an equivalence operator, as defined above. • Identity should imply equality, as stated earlier. • If either (or both) operand is a boxed value, equality should be computed by first unboxing any boxed operand(s), and then • applying the usual rules for equality on the resulting values. 	I	<u>7.2.5.2</u>	Equality
Equality of values	The values stored in the variables are equal if the sequences of characters are the same.	I	<u>7.2.5</u>	Identity and Equality of Values
Evaluation stack	Associated with each method state is an evaluation stack... The evaluation stack is made up of slots that can hold any data type, including an unboxed instance of a	I	<u>11.3.2.1</u>	The Evaluation Stack

	value type.			
Event contract	An event contract is specified with an event definition. There is an extensible set of operations for managing a named event, which includes three standard methods (register interest in an event, revoke interest in an event, fire the event). An event contract specifies method contracts for all of the operations that shall be implemented by any type that supports the event contract.	I	<u>7.6</u>	Contracts
Event definitions	The CTS supports events in precisely the same way that it supports properties... The conventional methods, however, are different and include means for subscribing and unsubscribing to events as well as for firing the event.	I	<u>7.11.4</u>	Event Definitions
Exception handling	Exception handling is supported in the CLI through exception objects and protected blocks of code	I	<u>11.4.2</u>	Exception Handling
Extended Array Library	This Library is not part of any Profile, but can be supplied as part of any CLI implementation. It provides support for non-vector arrays.	IV	<u>5.7</u>	Extended Array Library
Extended Numerics Library	The Extended Numerics Library is not part of any Profile, but can be supplied as part of any CLI implementation. It provides the support for floating-point (System.Float, System.Double) and extended-precision (System.Decimal) data types.	IV	<u>5.6</u>	Extended Numerics Library
Family accessibility	accessible to referents that support the same type, i.e. an exact type and all of the types that inherit from it	I	<u>7.5.3.2</u>	Accessibility of Members
Family-and-assembly accessibility	Accessible only to referents that qualify for both Family and Assembly access.	I	<u>7.5.3.2</u>	Accessibility of Members
Family-or-assembly accessibility	accessible only to referents that qualify for either Family or Assembly access.	I	<u>7.5.3.2</u>	Accessibility of Members
Field definitions	Field definitions name and a location signature.	I	<u>7.11.2</u>	Field Definitions
Field inheritance	A derived object type inherits all of the non-static fields of its base object type.	I	<u>7.10.1</u>	Field Inheritance
Fields	Fields are typed memory locations that store the data of a program.	II	<u>15</u>	Defining and Referencing Fields
File Names	A file name is like any other name where “.” is considered a normal constituent character. The specific syntax for file names follows the specifications of the underlying operating system	II	<u>5.8</u>	File Names
Finalizers	A class definition that creates an object type may supply an instance method to be called when an instance of the class is no longer accessible.	I	<u>7.9.6.7</u>	Finalizers
Getter method	By convention, properties define a getter method (for accessing the current value of the property)...	I	<u>7.11.3</u>	Property Definitions

Global Fields	In addition to types with static members, many languages have the notion of data and methods that are not part of a type at all. These are referred to as <i>global</i> fields and methods.	II	<u>9.8</u>	Global Fields and Methods
Global Methods	In addition to types with static members, many languages have the notion of data and methods that are not part of a type at all. These are referred to as <i>global</i> fields and methods.	II	<u>9.8</u>	Global Fields and Methods
Global state	The CLI manages multiple concurrent threads of control ... multiple managed heaps, and a shared memory address space.	I	<u>11.3.1</u>	The Global State
GUID	[A unique identification string used with remote procedure calls.]			
hide-by-name	The introduction of a name in a given type hides all inherited members of the same kind (method or field) with the same name.	II	<u>8.3</u>	Hiding
hide-by-name-and-sig	The introduction of a name in a given type hides any inherited member of the same kind but with precisely the same type (for fields) or signature (for methods, properties, and events).	II	<u>8.3</u>	Hiding
Hiding	Hiding controls which method names inherited from a base type are available for compile-time name binding.	II	<u>8</u>	Visibility, Accessibility and Hiding
Homes	The home of a data value is where it is stored for possible reuse	I	<u>11.1.6.1</u>	Homes for Values
Identifiers	Identifiers are used to name entities	II	<u>5.3</u>	Identifiers
Identity	<p>The identity operator is defined by the CTS as follows.</p> <ul style="list-style-type: none"> • If the values have different exact types, then they are not identical. • Otherwise, if their exact type is a Value Type, then they are identical if and only if the bit sequences of the values are the same, bit by bit. <p>Otherwise, if their exact type is a Reference Type, then they are identical if and only if the locations of the values are the same.</p>	I	<u>7.2.5.1</u>	Identity
Identity of values	The values of the variables are identical if the locations of the sequences of characters are the same, i.e., there is in fact only one string in memory.	I	<u>7.2.5</u>	Identity and Equality of Values
Ilasm	An assembler language for CIL	II	<u>2</u>	Overview
Inheritance demand	When attached to a type ..[an inheritance demand] requires that any type that wishes to inherit from this type shall have the specified security permission. When attached to a non-final virtual method it requires that any type that wishes to override this method shall have the specified permission.	I	<u>7.5.3.3</u>	Security Permissions
Instance Methods	Instance methods are associated with an instance of a type: within the body of an instance method it is possible	II	<u>14.2</u>	Static, Instance, and Virtual

Methods	to reference the particular instance on which the method is operating (via the <i>this</i> pointer).			Methods
Instruction pointer (IP)	An instruction pointer (IP) points to the next CIL instruction to be executed by the CLI in the present method.	I	<u>11.3.2</u>	Method State
Interface contract	Interface contracts specify which other contracts the interface supports, e.g. which interfaces, methods, properties and events shall be implemented.	I	<u>7.6</u>	Contracts
Interface type definition	An interface definition defines an interface type. An interface type is a named group of methods, locations and other contracts that shall be implemented by any object type that supports the interface contract of the same name.	I	<u>7.9.4</u>	Interface Type Definition
Interface type inheritance	Interface types may inherit from multiple interface types, i.e. an interface contract may list other interface contracts that shall also be supported.	I	<u>7.9.11</u>	Interface Type Inheritance
Interface types	Interface types describe a subset of the operations and none of the representation, and hence, cannot be an exact type of any value.	I	<u>7.2.3</u>	Classes, Interfaces and Objects
Interfaces	Interfaces...define a contract that other types may implement.	II	<u>11</u>	Semantics of Interfaces
Kernel Profile	This profile is the minimal possible conforming implementation of the CLI.	IV	<u>3.1</u>	The Kernel Profile
Labels	Provided as a programming convenience; they represent a number that is encoded in the metadata. The value represented by a label is typically an offset in bytes from the beginning of the current method, although the precise encoding differs depending on where in the logical metadata structure or CIL stream the label occurs.	II	<u>5.4</u>	Labels and Lists of Labels
Libraries	To a programmer a Library is a self-consistent set of types (classes, interfaces, and value types) that provide a useful set of functionality.	IV	<u>2.1</u>	Libraries
Local memory pool	The local memory pool is used to allocate objects whose type or size is not known at compile time and which the programmer does not wish to allocate in the managed heap.	I	<u>11.3.2.4</u>	Local Memory Pool
Local signatures	A local signature specifies the contract on a local variable allocated during the running of a method.	I	<u>7.6.1.3</u>	Local Signatures
Location signatures	All locations are typed. This means that all locations have a location signature , which defines constraints on the location, its usage, and on the usage of the values stored in the location.	I	<u>7.6.1.2</u>	Location Signatures
Locations	Values are stored in locations . A location can hold a single value at a time. All locations are typed. The type of the location embodies the requirements that shall be met by values that are stored in the location.	I	<u>7.3</u>	Locations
Machine state	One of the design goals of the CLI is to hide the details of a method call frame from the CIL code generator. <small>The machine state definitions reflect these decisions.</small>	I	<u>11.3</u>	Machine State

	The machine state definitions ... reflect these design choices, where machine state consists primarily of global state and method state.			
Managed code	Managed code is simply code that provides enough information to allow the CLI to provide a set of core services, including <ul style="list-style-type: none"> Given an address inside the code for a method, locate the metadata describing the method Walk the stack Handle exceptions Store and retrieve security information 	I	<u>5.2.1</u>	Managed Code
Managed data	Managed data is data that is allocated and released automatically by the CLI, through a process called garbage collection . Only managed code can access managed data, but programs that are written in managed code can access both managed and unmanaged data.	I	<u>5.2.2</u>	Managed Data
Managed pointer types	[The O and &] datatype represents an object reference that is managed by the CLI	I	<u>11.1.1.2</u>	Managed Pointer Types: O and &
Managed Pointers	Managed pointers (&) may point to a field of an object, a field of a value type, an element of an array, or the address where an element just past the end of an array would be stored (for pointer indexes into managed arrays).	II	<u>13.4.2</u>	Managed Pointers
Manifest	An <i>assembly</i> is a set of one or more files deployed as a unit.	II	<u>6</u>	Assemblies, Manifests and Modules
Marshalling Descriptors	A Marshalling Descriptor is like a signature – it's a blob of binary data. It describes how a field or parameter (which, as usual, covers the method return, as parameter number 0) should be marshalled when calling to or from unmanaged code via PInvoke dispatch or IJW ("It Just Works") thunking.	II	<u>22.4</u>	Marshaling Descriptors
Member	Fields, array elements, and methods are called members of the type. Properties and events are also members of the type.	I	<u>7.4</u>	Type Members
Member inheritance	Only object types may inherit implementations, hence only object types may inherit members	I	<u>7.10</u>	Member Inheritance
Memory store	By "memory store" we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes.	I	<u>11.6.1</u>	The Memory Store
Metadata	The CLI uses metadata to describe and reference the types defined by the Common Type System. Metadata is stored ("persisted") in a way that is independent of any particular programming language. Thus, metadata provides a common interchange mechanism for use	I	<u>5</u>	Overview of the Common Language Infrastructure

	between tools that manipulate programs (compilers, debuggers, etc.) as well as between these tools and the Virtual Execution System			
Metadata Token	This is a 4-byte value, that specifies a row in a metadata table, or a starting byte offset in the User String heap	III	<u>1.9</u>	Metadata Tokens
Method	A named method describes an operation that may be performed on values of an exact type.	I	<u>7.2.3</u>	Classes, Interfaces and Objects
Method contract	A method contract is specified with a method definition. A method contract is a named operation that specifies the contract between the implementation(s) of the method and the callers of the method.	I	<u>7.6</u>	Contracts
Method definitions	Method definitions are composed of a name, a method signature, and optionally an implementation of the method.	I	<u>7.11.1</u>	Method Definitions
Method inheritance	A derived object type inherits all of the instance and virtual methods of its base object type. It does not inherit constructors or static methods.	I	<u>7.10.2</u>	Method Inheritance
Method Pointers	Variables of type method pointer shall store the address of the entry point to a method with compatible signature.	II	<u>13.5</u>	Method Pointers
Method signatures	Method signatures are composed of <ul style="list-style-type: none"> • a calling convention, • a list of zero or more parameter signatures, one for each parameter of the method, • and a type signature for the result value if one is produced. 	I	<u>7.6.1.5</u>	Method Signatures
Method state	Method state describes the environment within which a method executes. (In conventional compiler terminology, it corresponds to a superset of the information captured in the "invocation stack frame").	I	<u>11.3.2</u>	Method State
methodInfo handle	This .. holds the signature of the method, the types of its local variables, and data about its exception handlers.	I	<u>11.3.2</u>	Method State
Module	A single file containing executable content	II	<u>6</u>	Assemblies, Manifests and Modules
Name Mangling the platform may use name-mangling rules that force the name as it appears to a managed program to differ from the name as seen in the native implementation (this is common, for example, when the native code is generated by a C++ compiler).	II	<u>14.5.2</u>	Platform Invoke
Native Data Types	Some implementations of the CLI will be hosted on top of existing operating systems or runtime platforms that specify data types required to perform certain functions. The metadata allows interaction with these <i>native data types</i> by specifying how the built-in and user-defined types of the CLI are to be marshalled to and from native	II	<u>7.4</u>	Native Data Types

	data types.			
Native size types	The native-size, or generic, types (I, U, O, and &) are a mechanism in the CLI for deferring the choice of a value's size.	I	<u>11.1.1</u>	Native Size: native int, native unsigned int, O and &
Nested type definitions	A nested type definition is identical to a top-level type definition, with one exception: a top-level type has a visibility attribute, while the visibility of a nested type is the same as the visibility of the enclosing type.	I	<u>7.11.5</u>	Nested Type Definitions
Nested types	A type (called a nested type) can be a member of an enclosing type.	I	<u>7.5.3.4</u>	Nested Types
Network Library	This Library is part of the Compact Profile. It provides simple networking services including direct access to network ports as well as HTTP support.	IV	<u>5.3</u>	Network Library
OOP	[Object Oriented Programming]			
Object type	The object type describes the physical structure of the instance and the operations that are allowed on it.	I	<u>7.9.6</u>	Object Type Definitions
Object type inheritance	With the sole exception of System.Object , which does not inherit from any other object type, all object types shall either explicitly or implicitly declare support for (inherit from) exactly one other object type.	I	<u>7.9.9</u>	Object Type Inheritance
Objects	Each object is self-typing, that is, its type is explicitly stored in its representation. It has an identity that distinguishes it from all other objects, and it has slots that store other entities (which may be either objects or values). While the contents of its slots may be changed, the identity of an object never changes.	I	<u>7</u>	Common Type System
Opaque classes	Some languages provide multi-byte data structures whose contents are manipulated directly by address arithmetic and indirection operations. To support this feature, the CLI allows value types to be created with a specified size but no information about their data members.	I	<u>11.1.6.3</u>	Opaque Classes
Overloading	Within a single scope, a given name may refer to any number of methods provided they differ in any of the following: Number of parameters [and] Type of each argument	I	<u>9.2</u>	Overloading
Overriding	Overriding deals with object layout and is applicable only to instance fields and virtual methods. The CTS provides two forms of member overriding, new slot and expect existing slot .	I	<u>7.10.4</u>	Hiding, Overriding, and Layout
Parameter	[Name used within the body of a method to refer to the corresponding argument of the method]			
Parameter passing	The CLI supports three kinds of parameter passing, all indicated in metadata as part of the signature of the method. Each parameter to a method has its own passing convention (e.g., the first parameter may be passed by-value while all others are passed by-ref).	I	<u>11.4.1.5</u>	Parameter Passing

Parameter Signatures	Parameter signatures define constraints on how an individual value is passed as part of a method invocation.	I	<u>7.6.1.4</u>	Parameter Signatures
Pinned	While a method with a pinned local variable is executing the VES shall not relocate the object to which the local refers.	II	<u>7.1.2</u>	Pinned
PInvoke	Methods defined in native code may be invoked using the platform invoke (also known as PInvoke or p/invoke) functionality of the CLI.	II	<u>14.5.2</u>	Platform Invoke
Pointer type	A pointer type is a compile time description of a value whose representation is a machine address of a location.	I	<u>7.2.1</u>	Value Types and Reference Types
Pointers	Pointers may contain the address of a field (of an object or value type) or an element of an array.	II	<u>13.4</u>	Pointer Types
Private accessibility	Accessible only to referents in the implementation of the exact type that defines the member.	I	<u>7.5.3.2</u>	Accessibility of Members
Profiles	A Profile is simply a set of Libraries, grouped together to form a consistent whole that provides a fixed level of functionality.	IV	<u>2.2</u>	Profiles
Properties	Property[ies] define named groups of accessor method definitions that implement the named event or property behavior.	I	<u>7.11</u>	Member Definitions
Property contract	A property contract is specified with a property definition. There is an extensible set of operations for handling a named value, which includes a standard pair for reading the value and changing the value. A property contract specifies method contracts for the subset of these operations that shall be implemented by any type that supports the property contract.	I	<u>7.6</u>	Contracts
Property definitions	A property definition defines a named value and the methods that access the value. A property definition defines the accessing contracts on that value.	I	<u>7.11.3</u>	Property Definitions
Public accessibility	Accessible to all referents	I	<u>7.5.3.2</u>	Accessibility of Members
Qualified name	...Consider a compound type Point that has a field named x . The name "field x " by itself does not uniquely identify the named field, but the qualified name "field x in type Point " does.	I	<u>7.5.2</u>	Assemblies and Scoping
Rank	The rank of an array is the number of dimensions.	II	<u>13.2</u>	Arrays
Reference demand	Any attempt to resolve a reference to the marked item shall have specified security permission.	I	<u>7.5.3.3</u>	Security Permissions
Reference types	Reference Types describe values that are represented as the location of a sequence of bits. There are three kinds of Reference Types:	I	<u>7.2.1</u>	Value Types and Reference Types
Reflection Library	This Library is part of the Compact Profile. It provides the ability to examine the structure of types, create instances of types, and invoke methods on types, all	IV	<u>5.4</u>	Reflection Library

	based on a description of the type.			
Remoting boundary	A remoting boundary exists if it is not possible to share the identity of an object directly across the boundary. For example, if two objects exist on physically separate machines that do not share a common address space, then a remoting boundary will exist between them.	I	<u>11.5</u>	Proxies and Remoting
Return state handle	This handle is used to restore the method state on return from the current method.	I	<u>11.3.2</u>	Method State
Runtime Infrastructure Library	This Library is part of the Kernel Profile. It provides the services needed by a compiler to target the CLI and the facilities needed to dynamically load types from a stream in the file format.	IV	<u>5.1</u>	Runtime Infrastructure Library
Scopes	Names are collected into groupings called scopes .	I	<u>7.5.2</u>	Assemblies and Scoping
Sealed	Specifies that a type shall not have subclasses	II	<u>9.1.4</u>	Inheritance Attributes
Sealed type	An object type declares it shall not be used as a base type (be inherited from) by declaring that it is a sealed type.	I	<u>7.9.9</u>	Object Type Inheritance
Security descriptor	This descriptor is not directly accessible to managed code but is used by the CLI security system to record security overrides (assert , permit-only , and deny).	I	<u>11.3.2</u>	Method State
Security permissions	Access to members is also controlled by security demands that may be attached to an assembly, type, method, property, or event.	I	<u>7.5.3.3</u>	Security Permissions
Serializable fields	A field that is marked serializable is to be serialized as part of the persistent state of a value of the type.	I	<u>7.11.2</u>	Field Definitions
Setter method	By convention, properties define ...optionally a setter method (for modifying the current value of the property).	I	<u>7.11.3</u>	Property Definitions
Signatures	Signatures are the part of a contract that can be checked and automatically enforced. Signatures are formed by adding constraints to types and other signatures.	I	<u>7.6.1</u>	Signatures
Simple labels	A simple label is a special name that represents an address	II	<u>5.4</u>	Labels and Lists of Labels
Special members	There are three special members, all methods, that can be defined as part of a type: instance constructors, instance finalizers, and type initializers.	II	<u>9.5</u>	Special Members
Special Types	Special Types are those that are referenced from CIL, but for which no definition is supplied: the VES supplies the definitions automatically based on information available from the reference.	II	<u>13</u>	Semantics of Special Types
Standard Profiles	There are two Standard Profiles. The smallest conforming implementation of the CLI is the Kernel Profile, while the Compact Profile contains additional features useful for applications targeting a more resource-rich set of devices.	IV	<u>3</u>	The Standard Profiles

Static fields	Types may declare locations that are associated with the type rather than any particular value of the type. Such locations are static fields of the type.	I	<u>7.4.3</u>	Static Fields and Static Methods
Static methods	...Types may also declare methods that are associated with the type rather than with values of the type. Such methods are static methods of the type.	I	<u>7.4.3</u>	Static Fields and Static Methods
Super Calls	In some cases, it may be desirable to re-use code defined in the base type. E.g., an overriding virtual method may want to call its previous version. This kind of re-use is called a <i>super call</i> , since the overridden method of the base type is called.			
This	When they are invoked, instance and virtual methods are passed the value on which this invocation is to operate (known as this or a this pointer).	I	<u>7.4.2</u>	Methods
Thunk	A (typically) small piece of code used to provide a transition between two pieces of code where special handling is required			
Try block	In the CLI, a method may define a range of CIL instructions that are said to be <i>protected</i> . This is called the try block .	II	<u>18</u>	Exception Handling
Type definers	Type definers construct a new type from existing types.	I	<u>7.9</u>	Type Definers
Type definition	<p>The type definition:</p> <ul style="list-style-type: none"> Defines a name for the type being defined, i.e. the type name, and specifies a scope in which that name will be found Defines a member scope in which the names of the different kinds of members (fields, methods, events, and properties) are bound. The tuple of (member name, member kind, and member signature) is unique within a member scope of a type. Implicitly assigns the type to the assembly scope of the assembly that contains the type definition. 	I	<u>7.5.2</u>	Assemblies and Scoping
Type inheritance	Inheritance of types is another way of saying that the derived type guarantees support for all of the type contracts of the base type. In addition, the derived type usually provides additional functionality or specialized behavior.	I	<u>7.9.8</u>	Type Inheritance
Type members	Object type definitions include member definitions for all of the members of the type. Briefly, members of a type include fields into which values are stored, methods that may be invoked, properties that are available, and events that may be raised.	I	<u>7.4</u>	Type Members
Type safety	An implementation that lives up to the enforceable part of the contract (the named signatures) is said to be typesafe .	I	<u>7.8</u>	Type Safety and Verification

Type signatures	Type signatures define the constraints on a value and its usage.	I	<u>7.6.1.1</u>	Type Signatures
Typed reference parameters	A runtime representation of the data type is passed along with the address of the data, and the type of the parameter is therefore one specially supplied for this purpose.	I	<u>11.4.1.5</u>	Parameter Passing
Types	Types describe values. All places where values are stored, passed, or operated upon have a type, e.g. all variables, parameters, evaluation stack locations, and method results. The type defines the allowable values and the allowable operations supported by the values of the type. All operators and functions have expected types for each of the values accessed or used.	I	<u>7.2</u>	Values and Types
Unary operators	Unary operators take one argument, perform some operation on it, and return the result. They are represented as static methods on the class that defines the type of their one operand or their return type.	I	<u>9.3.1</u>	Unary Operators
Unbox	Unbox is a narrowing (runtime exception may be generated) operation that converts a <code>System.Object</code> (whose runtime type is a value type) to a value type instance.	I	<u>11.1.6.2.5</u>	Boxing and Unboxing
Unmanaged Code	[Code that does not require the runtime for execution. This code may not use the common type system or other features of the runtime. Traditional native code (before the CLI) is considered unmanaged]			
Unmanaged pointer types	An unmanaged pointer type (also known simply as a "pointer type") is defined by specifying a location signature for the location the pointer references. Any signature of a pointer type includes this location signature.	I	<u>7.9.2</u>	Unmanaged Pointer Types
Validation	Validation refers to a set of tests that can be performed on any file to check that the file format, metadata, and CLI are self-consistent.	II	<u>3</u>	Validation and Verification
Value type inheritance	Value Types, in their unboxed form, do not inherit from any type.	I	<u>7.9.10</u>	Value Type inheritance
Value types	In contrast to classes, value types (see Partition I) are not accessed by using a reference but are stored directly in the location of that type.	II	<u>12</u>	Semantics of Value Types
Values	The representation of a value (except for those of built-in types) can be subdivided into sub-values. These sub-values are either named, in which case they are called fields , or they are accessed by an indexing expression, in which case they are called array elements .	I	<u>7.4.1</u>	Fields, Array Elements, and Values
Vararg Methods	vararg methods accept a variable number of arguments.	II	<u>14.4.5</u>	Vararg methods
Variable argument lists	The CLI works in conjunction with the class library to implement methods that accept argument lists of unknown length and type ("varargs methods").	I	<u>11.3.2.3</u>	Variable Argument Lists

Vectors	Vectors are single-dimension arrays with a zero lower bound.	II	<u>13.1</u>	Vectors
Verifiability	Memory safety is a property that ensures programs running in the same address space are correctly isolated from one another ... Thus, it is desirable to test whether programs are memory safe prior to running them. Unfortunately, it is provably impossible to do this with 100% accuracy. Instead, the CLI can test a stronger restriction, called <i>verifiability</i> .	III	<u>1.8</u>	Verifiability
Verification	<i>Verification</i> refers to a check of both CIL and its related metadata to ensure that the CIL code sequences do not permit any access to memory outside the program's logical address space.	II	<u>3</u>	Validation and Verification
Version Number	The version number of the assembly, specified as four 32-bit integers	II	<u>6.2.1.4</u>	Version Numbers
Virtual call	...A virtual method may be invoked by a special mechanism (a <i>virtual call</i>) that chooses the implementation based on the dynamically detected type of the instance used to make the virtual call rather than the type statically known at compile time.	I	<u>7.4.4</u>	Virtual Methods
Virtual calling convention	The CIL provides a "virtual calling convention" that is converted by an interpreter or JIT compiler into a native calling convention.	I	<u>11.4.1.4</u>	Virtual Calling Convention
Virtual execution system	The Virtual Execution System (VES) provides an environment for executing managed code. It provides direct support for a set of built-in data types, defines a hypothetical machine with an associated machine model and state, a set of control flow constructs, and an exception handling model.	I	<u>5</u>	Overview of the Common Language Infrastructure
Virtual methods	Virtual methods are associated with an instance of a type in much the same way as for instance methods. However, unlike instance methods, it is possible to call a virtual method in such a way that the implementation of the method shall be chosen at runtime by the VES depends upon the type of object used for the <i>this</i> pointer.	II	<u>14.2</u>	Static, Instance, and Virtual Methods
Visibility	Attached only to top-level types, and there are only two possibilities: visible to types within the same assembly, or visible to types regardless of assembly.	II	<u>8.1</u>	Visibility of Top-Level Types and Accessibility of Nested Types
Widen	If a type overrides an inherited method, it may <i>widen</i> , but it shall not <i>narrow</i> , the accessibility of that method.	II	<u>9.3.3</u>	Accessibility and Overriding
XML Library	This Library is part of the Compact Profile. It provides a simple "pull-style" parser for XML. It is designed for resource-constrained devices, yet provides a simple user model.	IV	<u>5.5</u>	XML Library

1 5 Overview of the Common Language Infrastructure

2 The Common Language Infrastructure (CLI) provides a specification for executable code and the execution
3 environment (the Virtual Execution System, or VES) in which it runs. Executable code is presented to the VES
4 as **modules**. A module is a single file containing executable content in the format specified in [Partition II](#).

5 The remainder of this section and its subsections contain only informative text

6 At the center of the Common Language Infrastructure (CLI) is a single type system, the Common Type System
7 (CTS), that is shared by compilers, tools, and the CLI itself. It is the model that defines the rules the CLI
8 follows when declaring, using, and managing types. The CTS establishes a framework that enables cross-
9 language integration, type safety, and high performance code execution. This section describes the architecture
10 of CLI by describing the CTS.

11 The following four areas are covered in this section:

- 12 • **The Common Type System.** See [Chapter 7](#). The Common Type System (CTS) provides a rich
13 type system that supports the types and operations found in many programming languages. The
14 Common Type System is intended to support the complete implementation of a wide range of
15 programming languages.
- 16 • **Metadata.** See [Chapter 8](#). The CLI uses metadata to describe and reference the types defined by
17 the Common Type System. Metadata is stored ("persisted") in a way that is independent of any
18 particular programming language. Thus, metadata provides a common interchange mechanism for
19 use between tools that manipulate programs (compilers, debuggers, etc.) as well as between these
20 tools and the Virtual Execution System.
- 21 • **The Common Language Specification.** See [Chapter 9](#). The Common Language Specification is
22 an agreement between language designers and framework (class library) designers. It specifies a
23 subset of the CTS Type System and a set of usage conventions. Languages provide their users the
24 greatest ability to access frameworks by implementing at least those parts of the CTS that are part
25 of the CLS. Similarly, frameworks will be most widely used if their publicly exposed aspects
26 (classes, interfaces, methods, fields, etc.) use only types that are part of the CLS and adhere to the
27 CLS conventions.
- 28 • **The Virtual Execution System.** See [Chapter 11](#). The Virtual Execution System (VES)
29 implements and enforces the CTS model. The VES is responsible for loading and running
30 programs written for the CLI. It provides the services needed to execute managed code and data,
31 using the metadata to connect separately generated modules together at runtime (late binding).

32 Together, these aspects of the CLI form a unifying framework for designing, developing, deploying, and
33 executing distributed components and applications. The appropriate subset of the Common Type System is
34 available from each programming language that targets the CLI. Language-based tools communicate with each
35 other and with the Virtual Execution System using metadata to define and reference the types used to construct
36 the application. The Virtual Execution System uses the metadata to create instances of the types as needed and
37 to provide data type information to other parts of the infrastructure (such as remoting services, assembly
38 downloading, security, etc.).

39 5.1 Relationship to Type Safety

40 Type safety is usually discussed in terms of what it does, e.g. guaranteeing encapsulation between different
41 objects, or in terms of what it prevents, e.g. memory corruption by writing where one shouldn't. However, from
42 the point of view of the Common Type System, type safety guarantees that:

- 43 • **References are what they say they are** - Every reference is typed and the object or value
44 referenced also has a type, and they are assignment compatible (see [Section 7.7](#)).
- 45 • **Identities are who they say they are** - There is no way to corrupt or spoof an object, and by
46 implication a user or security domain. The access to an object is through accessible functions and
47 fields. An object may still be designed in such a way that security is compromised. However, a

1 local analysis of the class, its methods, and the things it uses, as opposed to a global analysis of
2 all uses of a class, is sufficient to assess the vulnerabilities.

3 • **Only appropriate operations can be invoked** – The reference type defines the accessible
4 functions and fields. This includes limiting visibility based on where the reference is, e.g.
5 protected fields only visible in subclasses.

6 The Common Type System promotes type safety e.g. everything is typed. Type safety can be optionally
7 enforced. The hard problem is determining if an implementation conforms to a typesafe declaration. Since the
8 declarations are carried along as metadata with the compiled form of the program, a compiler from the
9 Common Intermediate Language (CIL) to native code (see [Section 7.8](#)) can type-check the implementations.

10 5.2 Relationship to Managed Metadata-driven Execution

11 Metadata describes code by describing the types that the code defines and the types that it references externally.
12 The compiler produces the metadata when the code is produced. Enough information is stored in the metadata
13 to:

14 • **Manage code execution** – not just load and execute, but also memory management and execution
15 state inspection.
16 • **Administer the code** – Installation, resolution, and other services
17 • **Reference types in the code** – Importing into other languages and tools as well as scripting and
18 automation support.

19 The Common Type System assumes that the execution environment is metadata-driven. Using metadata allows
20 the CLI to support:

21 • **Multiple execution models** - The metadata also allows the execution environment to deal with a
22 mixture of interpreted, JITted, native and legacy code and still present uniform services to tools
23 like debuggers or profilers, consistent exception handling and unwinding, reliable code access
24 security, and efficient memory management.
25 • **Auto support for services** - Since the metadata is available at execution time, the execution
26 environment and the base libraries can automatically supply support for reflection, automation,
27 serialization, remote objects, and inter-operability with existing unmanaged native code with little
28 or no effort on the part of the programmer.
29 • **Better optimization** – Using metadata references instead of physical offsets, layouts, and sizes
30 allows the CLI to optimize the physical layouts of members and dispatch tables. In addition, this
31 allows the generated code to be optimized to match the particular CPU or environment.
32 • **Reduced binding brittleness** – Using metadata references reduces version-to-version brittleness
33 by replacing compile-time object layout with load-time layout and binding by name.
34 • **Flexible deployment resolution** - Since we can have metadata for both the reference and the
35 definition of a type, more robust and flexible deployment and resolution mechanisms are possible.
36 Resolution means that by looking in the appropriate set of places it is possible to find the
37 implementation that best satisfies these requirements for use in this context. There are five
38 elements of information in the foregoing: two items are made available via metadata
39 (requirements and context); the others come from application packaging and deployment (where
40 to look, how to find an implementation, and how to decide the best match).

41 5.2.1 Managed Code

42 Managed code is simply code that provides enough information to allow the CLI to provide a set of core
43 services, including

44 • Given an address inside the code for a method, locate the metadata describing the method
45 • Walk the stack
46 • Handle exceptions

1 • Store and retrieve security information

2 This standard specifies a particular instruction set, the Common Intermediate Language (CIL, see Partition III),
3 and a file format (see Partition II) for storing and transmitting managed code.

4 **5.2.2 Managed Data**

5 **Managed data** is data that is allocated and released automatically by the CLI, through a process called
6 **garbage collection**.

7 **5.2.3 Summary**

8 The Common Type System is about integration between languages: using another language's objects as if they
9 were one's own.

10 The objective of the CLI is to make it easier to write components and applications from any language. It does
11 this by defining a standard set of types, making all components fully self-describing, and providing a high
12 performance common execution environment. This ensures that all CLI compliant system services and
13 components will be accessible to all CLI aware languages and tools. In addition, this simplifies deployment of
14 components and applications that use them, all in a way that allows compilers and other tools to leverage the
15 high performance execution environment. The Common Type System covers, at a high level, the concepts and
16 interactions that make all of this possible.

17 The discussion is broken down into four areas:

18 • Type System – What types are and how to define them.
19 • Metadata – How types are described and how those descriptions are stored.
20 • Common Language Specification – Restrictions required for language interoperability.
21 • Virtual Execution System – How code is executed and types are instantiated, interact, and die.

22 **End informative text**

1 6 Common Language Specification (CLS)

2 6.1 Introduction

3 The Common Language Specification (CLS) is a set of rules intended to promote language interoperability.
4 These rules shall be followed in order to conform to the CLS. They are described in greater detail in
5 subsequent chapters and are summarized in [Chapter 10](#). CLS conformance is a characteristic of types that are
6 generated for execution on a CLI implementation. Such types must conform to the CLI specification, in
7 addition to the CLS rules. These additional rules apply only to types that are visible in assemblies other than
8 those in which they are defined, and to the members (fields, methods, properties, events, and nested types) that
9 are accessible outside the assembly (i.e. those that have an accessibility of **public**, **family**, or **family-or-**
10 **assembly**).

11 **Note:** A library consisting of CLS-compliant code is herein referred to as a “framework”. Compilers that
12 generate code for the CLI may be designed to make use of such libraries, but not to be able to produce or
13 extend such library code. These compilers are referred to as “consumers”. Compilers that are designed to both
14 produce and extend frameworks are referred to as “extenders”. In the description of each CLS rule, additional
15 informative text is provided to assist the reader in understanding the rule’s implication for each of these
16 situations.

17 6.2 Views of CLS Compliance

18 **This section and its subsections contain only informative text**

19 The CLS is a set of rules that apply to generated assemblies. Because the CLS is designed to support
20 interoperability for libraries and the high-level programming languages used to write them, it is often useful to
21 think of the CLS rules from the perspective of the high-level source code and tools, such as compilers, that are
22 used in the process of generating assemblies. For this reason, informative notes are added to the description of
23 CLS rules to assist the reader in understanding the rule’s implications for several different classes of tools and
24 users. The different viewpoints used in the description are called **framework**, **consumer**, and **extender** and are
25 described here.

26 6.2.1 CLS Framework

27 A library consisting of CLS-compliant code is herein referred to as a “framework”. Frameworks (libraries) are
28 designed for use by a wide range of programming languages and tools, including both CLS consumer and
29 extender languages. By adhering to the rules of the CLS, authors of libraries ensure that the libraries will be
30 usable by a larger class of tools than if they chose not to adhere to the CLS rules. The following are some
31 additional guidelines that CLS-compliant frameworks should follow:

- 32 • Avoid the use of names commonly used as keywords in programming languages
- 33 • Should not expect users of the framework to be able to author nested types
- 34 • Should assume that implementations of methods of the same name and signature on different
35 interfaces are independent.
- 36 • Should not rely on initialization of value types to be performed automatically based on specified
37 initializer values.

38 6.2.2 CLS Consumer

39 A CLS consumer is a language or tool that is designed to allow access to all of the features supplied by CLS-
40 compliant frameworks (libraries), but not necessarily be able to produce them. The following is a partial list of
41 things CLS consumer tools are expected to be able to do:

- 42 • Support calling any CLS-compliant method or delegate..
- 43 • Have a mechanism for calling methods that have names that are keywords in the language

- Support calling distinct methods supported by a type that have the same name and signature, but implement different interfaces
- Create an instance of any CLS-compliant type
- Read and modify any CLS-compliant field
- Access nested types
- Access any CLS-compliant property. This does not require any special support other than the ability to call the getter and setter methods of the property.
- Access any CLS-compliant event. This does not require any special support other than the ability to call methods defined for the event.

The following is a list of things CLS consumer tools need not support:

- Creation of new types or interfaces
- Initialization metadata (see [Partition II](#)) on fields and parameters other than static literal fields. Note that consumers may choose to use initialization metadata, but may also safely ignore such metadata on anything other than static literal fields.

6.2.3 CLS Extender

A CLS extender is a language or tool that is designed to allow programmers to both use and extend CLS-compliant frameworks. CLS extenders support a superset of the behavior supported by a CLS consumer, i.e., everything that applies to a CLS consumer also applies to CLS extenders. In addition to the requirements of a consumer, extenders are expected to be able to:

- Define new CLS-compliant types that extend any (non-sealed) CLS-compliant base class
- Have some mechanism for defining types with names that are keywords in the language
- Provide independent implementations for all methods of all interfaces supported by a type. That is, it is not sufficient for an extender to require a single code body to implement all interface methods of the same name and signature.
- Implement any CLS-compliant interface
- Place any CLS-compliant custom attribute on all appropriate elements of metadata

Extenders need not support the following:

- Definition of new CLS-compliant interfaces
- Definition of nested types

The common language specification is designed to be large enough that it is properly expressive and small enough that all languages can reasonably accommodate it.

End informative text

6.3 CLS Compliance

As these rules are introduced in detail, they are described in a common format. For an example, see the first rule below. The first paragraph specifies the rule itself. This is then followed by an informative description of the implications of the rule from the three different viewpoints as described above.

The CLS defines language interoperability rules, which apply only to "externally visible" items. The CLS unit of that language interoperability is the assembly—that is, within a single assembly there are no restrictions as to the programming techniques that are used. Thus, the CLS rules apply only to items that are visible (see [clause 7.5.3](#)) outside of their defining assembly and have public, family, or family-or-assembly accessibility (see [clause 7.5.3.2](#)).

1 **CLS Rule 1:** CLS rules apply only to those parts of a type that are accessible or visible outside of the defining
2 assembly.

3 **Note:**

4 **CLS (consumer):** no impact.

5 **CLS (extender):** when checking CLS compliance at compile time, be sure to apply the rules only to
6 information that will be exposed outside the assembly.

7 **CLS (framework):** CLS rules do not apply to internal implementation within an assembly. A type is **CLS-**
8 **compliant** if all its publicly accessible parts (those classes, interfaces, methods, fields, properties, and events
9 that are available to code executing in another assembly) either

- have signatures composed only of CLS-compliant types, or
- are specifically marked as not CLS-compliant

12 Any construct that would make it impossible to rapidly verify code is excluded from the CLS. This allows all
13 CLS-compliant languages to produce verifiable code if they so choose.

14 6.3.1 Marking Items as CLS-Compliant

15 The CLS specifies how to mark externally visible parts of an assembly to indicate whether or not they comply
16 with the CLS requirements. This is done using the custom attribute mechanism (see [Section 8.7](#) and
17 [Partition II](#)). The class `System.CLSCompliantAttribute` (see [Partition IV](#)) indicates which types and type
18 members are CLS-compliant. It also can be attached to an assembly, to specify the default value for all top-
19 level types it contains.

20 The constructor for `System.CLSCompliantAttribute` takes a Boolean argument indicating whether the item
21 with which it is associated is or is not CLS-compliant. This allows any item (assembly, type, or type member)
22 to be explicitly marked as CLS-compliant or not.

23 The rules for determining CLS compliance are:

- When an assembly does not carry an explicit `System.CLSCompliantAttribute`, it shall be assumed to carry `System.CLSCompliantAttribute(false)`.
- By default, a type inherits the CLS-compliance attribute of its enclosing type (for nested types) or acquires the value attached to its assembly (for top-level types). It may be marked as either CLS-compliant or not CLS-Compliant by attaching the `System.CLSCompliantAttribute` attribute.
- By default, other members (methods, fields, properties and events) inherit the CLS-compliance of their type. They may be marked as not CLS-compliant by attaching the attribute `System.CLSCompliantAttribute(false)`.

32 **CLS Rule 2:** Members of non-CLS compliant types shall not be marked CLS-compliant.

33 **Note:**

34 **CLS (consumer):** May ignore any member that is not CLS-compliant using the above rules.

35 **CLS (extender):** Should encourage correct labeling of newly authored assemblies, classes, interfaces, and
36 methods. Compile-time enforcement of the CLS rules is strongly encouraged.

37 **CLS (framework):** Shall correctly label all publicly exposed members as to their CLS compliance. The rules
38 specified here may be used to minimize the number of markers required (for example, label the entire assembly
39 if all types and members are compliant or if there are only a few exceptions that need to be marked).

1 7 **Common Type System**

2 Types describe values and specify a contract (see [Section 7.6](#)) that all values of that type shall support. Because
3 the CTS supports Object-Oriented Programming (OOP) as well as functional and procedural programming
4 languages, it deals with two kinds of entities: Objects and Values. Values are simple bit patterns for things like
5 integers and floats; each value has a type that describes both the storage that it occupies and the meanings of
6 the bits in its representation, and also the operations that may be performed on that representation. Values are
7 intended for representing the corresponding simple types in programming languages like C, and also for
8 representing non-objects in languages like C++ and Java™.

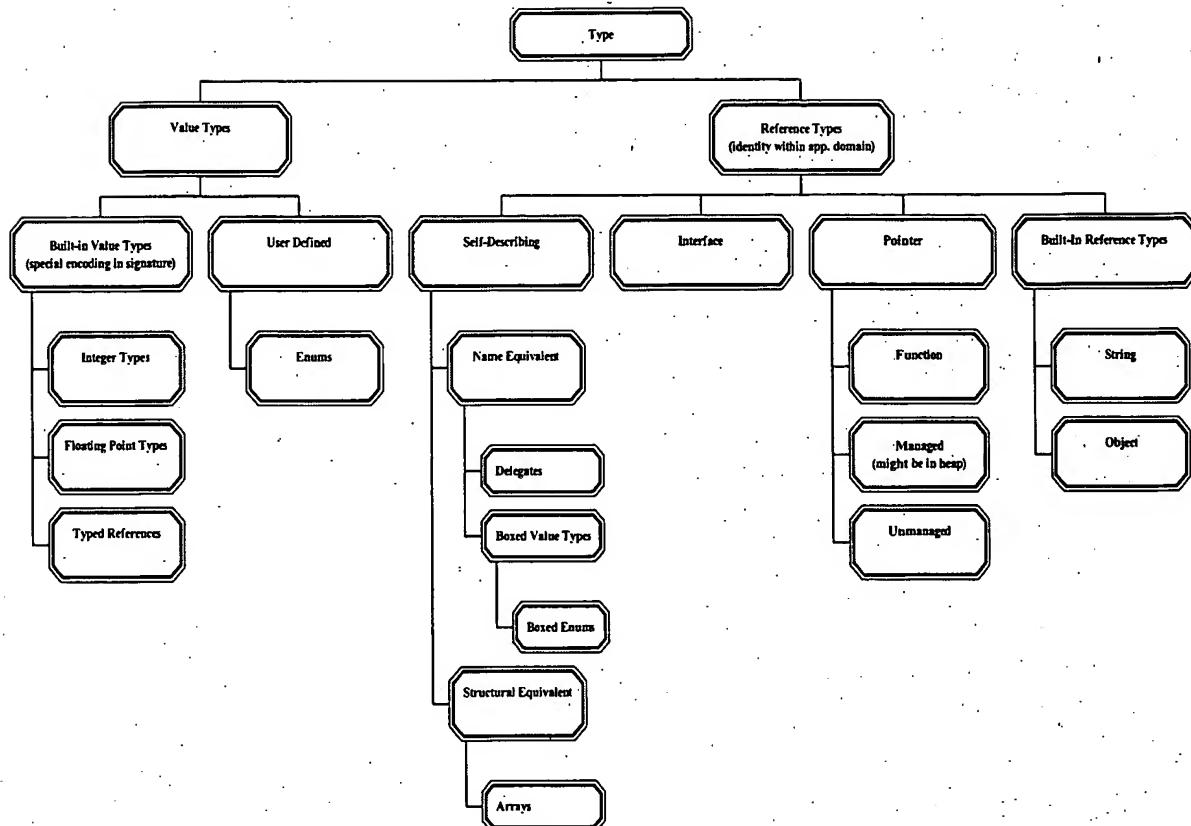
9 Objects have rather more to them than do values. Each object is self-typing, that is, its type is explicitly stored
10 in its representation. It has an identity that distinguishes it from all other objects, and it has slots that store other
11 entities (which may be either objects or values). While the contents of its slots may be changed, the identity of
12 an object never changes.

13 There are several kinds of Objects and Values, as shown in the following diagram.

14

1

Figure 1: Type System



2

3

4

1 7.1 Relationship to Object-Oriented Programming

2 This section contains only informative text

3 The term **type** is often used in the world of value-oriented programming to mean data representation. In the
4 object-oriented world it usually refers to behavior rather than to representation. In the CTS, type is used to
5 mean both of these things: two entities have the same type if and only if they have both compatible
6 representations and behaviors. Thus, in the CTS, if one type is derived from a base type, then instances of the
7 derived type may be substituted for instances of the base type because **both** the representation and the behavior
8 are compatible.

9 In the CTS, unlike some OOP languages, two objects that have fundamentally different representations have
10 different types. Some OOP languages use a different notion of type. They consider two objects to have the
11 same type if they respond in the same way to the same set of messages. This notion is captured in the CTS by
12 saying that the objects implement the same interface.

13 Similarly, some OOP languages (e.g. SmallTalk) consider message passing to be the fundamental model of
14 computation. In the CTS, this corresponds to calling virtual methods (see [clause 7.4.4](#)), where the signature of
15 the virtual method serves the role of the message.

16 The CTS itself does not directly capture the notion of “typeless programming.” That is, there is no way to call
17 a non-static method without knowing the type of the object. Nevertheless, typeless programming can be
18 implemented based on the facilities provided by the reflection package (see [Partition IV](#)) if it is implemented.

19 End informative text

20 7.2 Values and Types

21 Types describe values. All places where values are stored, passed, or operated upon have a type, e.g. all
22 variables, parameters, evaluation stack locations, and method results. The type defines the allowable values and
23 the allowable operations supported by the values of the type. All operators and functions have expected types
24 for each of the values accessed or used.

25 A value can be of more than one type. A value that supports many interfaces is an example of a value that is of
26 more than one type, as is a value that inherits from another.

27 7.2.1 Value Types and Reference Types

28 There are two kinds of types: **Value Types** and **Reference Types**.

- **Value Types** - Value Types describe values that are represented as sequences of bits.
- **Reference Types** - Reference Types describe values that are represented as the location of a sequence of bits. There are four kinds of Reference Types:
 - **An object type** is a reference type of a self-describing value (see [clause 7.2.3](#)). Some object types (e.g. abstract classes) are only a partial description of a value.
 - **An interface type** is always a partial description of a value, potentially supported by many object types.
 - **A pointer type** is a compile time description of a value whose representation is a machine address of a location.
 - **Built-in types**

39 7.2.2 Built-in Types

40 The following data types are an integral part of the CTS and are supported directly by the Virtual Execution
41 System (VES). They have special encoding in the persisted metadata:

Table 1: Special Encoding

Name in CIL assembler (see <u>Partition II</u>)	CLS Type?	Name in class library (see <u>Partition IV</u>)	Description
bool	Yes	System.Boolean	True/false value
char	Yes	System.Char	Unicode 16-bit char.
object	Yes	System.Object	Object or boxed value type
string	Yes	System.String	Unicode string
float32	Yes	System.Single	IEC 60559:1989 32-bit float
float64	Yes	System.Double	IEC 60559:1989 64-bit float
int8	No	System.SByte	Signed 8-bit integer
int16	Yes	System.Int16	Signed 16-bit integer
int32	Yes	System.Int32	Signed 32-bit integer
int64	Yes	System.Int64	Signed 64-bit integer
native int	Yes	System.IntPtr	Signed integer, native size
native unsigned int	No	System.UIntPtr	Unsigned integer, native size
typedref	No	System.TypedReference	Pointer plus runtime type
unsigned int8	Yes	System.Byte	Unsigned 8-bit integer
unsigned int16	No	System.UInt16	Unsigned 16-bit integer
unsigned int32	No	System.UInt32	Unsigned 32-bit integer
unsigned int64	No	System.UInt64	Unsigned 64-bit integer

7.2.3 Classes, Interfaces and Objects

Every value has an exact type that **fully describes** the value. A type fully describes a value if it completely defines the value's representation and the operations defined on the value.

For a Value Type, defining the representation entails describing the sequence of bits that make up the value's representation. For a Reference Type, defining the representation entails describing the location and the sequence of bits that make up the value's representation.

A **method** describes an operation that may be performed on values of an exact type. Defining the set of operations allowed on values of an exact type entails specifying named methods for each operation.

Some types are only a partial description, e.g. **interface types**. Interface types describe a subset of the operations and none of the representation, and hence, cannot be an exact type of any value. Hence, while a value has only one exact type, it may also be a value of many other types as well. Furthermore, since the exact type fully describes the value, it also fully specifies all of the other types that a value of the exact type can have.

While it is true that every value has an exact type, it is not always possible to determine the exact type by inspecting the representation of the value. In particular, it is *never* possible to determine the exact type of a value of a Value Type. Consider two of the built-in Value Types, 32-bit signed and unsigned integers. While each type is a full specification of their respective values, i.e. an exact type, there is no way to derive that exact type from a value's particular 32-bit sequence.

For some values, called **objects**, it is always possible to determine the exact type from the value. Exact types of objects are also called **object types**. Objects are values of Reference Types, but not all Reference Types describe objects. Consider a value that is a pointer to a 32-bit integer, a kind of Reference Type. There is no way to discover the type of the value by examining the pointer bits, hence it is not an object. Now consider the built-in CTS Reference Type System.String (see Partition IV). The exact type of a value of this type is always

1 determinable by examining the value, hence values of type `System.String` are objects and `System.String` is an
2 object type.

3 **7.2.4 Boxing and Unboxing of Values**

4 For every Value Type, the CTS defines a corresponding Reference Type called the **boxed type**. The reverse is
5 not true: Reference Types do not in general have a corresponding Value Type. The representation of a value of
6 a boxed type (a **boxed value**) is a location where a value of the Value Type may be stored. A boxed type is an
7 object type and a boxed value is an object.

8 All Value Types have an operation called **box**. Boxing a value of any Value Type produces its boxed value, i.e.
9 a value of the corresponding boxed type containing a bit copy of the original value. All boxed types have an
10 operation called **unbox**. Unboxing results in a managed pointer to the bit representation of the value.

11 Notice that interfaces and inheritance are defined only on Reference types. Thus, while a Value Type definition
12 (see [clause 7.9.7](#)) can specify both interfaces that shall be implemented by the Value Type and the class
13 (`System.ValueType` or `System.Enum`) from which it inherits, these apply only to boxed values.

14 **CLS Rule 3:** The CLS does not include boxed value types.

15 **Note:**

16 **In lieu of boxed types**, use `System.Object`, `System.ValueType` or `System.Enum`, as appropriate. (See

18 **CLS (consumer)**: need not import boxed value types.

19 **CLS (extender)**: need not provide syntax for defining or using boxed value types.

20 **CLS (framework)**: shall not use boxed value types in their publicly exposed aspects.

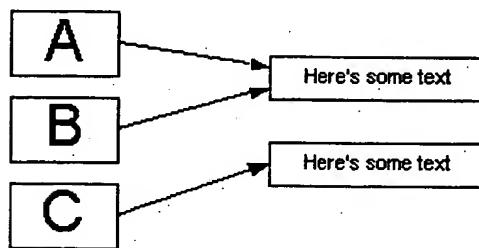
21 **7.2.5 Identity and Equality of Values**

22 There are two binary operators defined on all pairs of values, **identity** and **equality**, that return a Boolean
23 result. Both of these operators are mathematical **equivalence** operators, i.e. they are:

- Reflexive - $a \text{ op } a$ is true.
- Symmetric - $a \text{ op } b$ is true if and only if $b \text{ op } a$ is true.
- Transitive - if $a \text{ op } b$ is true and $b \text{ op } c$ is true, then $a \text{ op } c$ is true

27 In addition, identity always implies equality, but not the reverse, i.e., the equality operator need not be the same
28 as the identity operator as long as two identical values are also equal values.

29 To understand the difference between these operations, consider three variables whose type is `System.String`,
30 where the arrow is intended to mean "is a reference to":



31 The values of the variables are **identical** if the locations of the sequences of characters are the same, i.e., there
32 is in fact only one string in memory. The values stored in the variables are **equal** if the sequences of characters
33 are the same. Thus, the values of variables A and B are identical, the values of variables A and C as well as B
34 and C are not identical, and the values of all three of A, B, and C are equal.

1 7.2.5.1 **Identity**

2 The identity operator is defined by the CTS as follows.

3 • If the values have different exact types, then they are not identical.

4 • Otherwise, if their exact type is a Value Type, then they are identical if and only if the bit

5 sequences of the values are the same, bit by bit.

6 • Otherwise, if their exact type is a Reference Type, then they are identical if and only if the

7 locations of the values are the same.

8 Identity is implemented on `System.Object` via the `ReferenceEquals` method.

9 7.2.5.2 **Equality**

10 For value types, the equality operator is part of the definition of the exact type. Definitions of equality should
11 obey the following rules:

12 • Equality should be an equivalence operator, as defined above.

13 • Identity should imply equality, as stated earlier.

14 • If either (or both) operand is a boxed value, equality should be computed by

15 o first unboxing any boxed operand(s), and then

16 o applying the usual rules for equality on the resulting values.

17 Equality is implemented on `System.Object` via the `Equals` method.

18 Note: Although two floating point NaNs are defined by IEC 60559:1989 to always compare as unequal, the
19 contract for `System.Object.Equals`, requires that overrides must satisfy the requirements for an equivalence
20 operator. Therefore, `System.Double.Equals` and `System.Single.Equals` return `True` when comparing two
21 NaNs, while the equality operator returns `False` in that case, as required by the standard.

22 7.3 **Locations**

23 Values are stored in **locations**. A location can hold a single value at a time. All locations are typed. The type of
24 the location embodies the requirements that shall be met by values that are stored in the location. Examples of
25 locations are local variables and parameters.

26 More importantly, the type of the location specifies the restrictions on usage of any value that is loaded from
27 the location. For example, a location can hold values of potentially many exact types as long as all of the values
28 are assignment compatible with the type of the location (see below). All values loaded from a location are
29 treated as if they are of the type of the location. Only operations valid for the type of the location may be
30 invoked even if the exact type of the value stored in the location is capable of additional operations.

31 7.3.1 **Assignment Compatible Locations**

32 A value may be stored in a location only if one of the types of the value is **assignment compatible** with the
33 type of the location. A type is always assignment compatible with itself. Assignment compatibility can often be
34 determined at compile time, in which case there is no need for testing at run time. Assignment compatibility is
35 described in detail in [Section 7.7](#).

36 7.3.2 **Coercion**

37 Sometimes it is desirable to take a value of a type that is *not* assignment compatible with a location and convert
38 the value to a type that *is* assignment compatible. This is accomplished through **coercion** of the value. Coercion
39 takes a value of a particular type and a desired type and attempts to create a value of the desired type that has
40 equivalent meaning to the original value. Coercion can result in representation changes as well as type changes,
41 hence coercion does not necessarily preserve the identity of two objects.

42 There are two kinds of coercion: **widening**, which never loses information, and **narrowing**, in which
43 information may be lost. An example of a widening coercion would be coercing a value that is a 32-bit signed

1 integer to a value that is a 64-bit signed integer. An example of a narrowing coercion is the reverse; coercing a
2 64-bit signed integer to a 32-bit signed integer. Programming languages often implement widening coercions as
3 **implicit conversions**, whereas narrowing coercions usually require an **explicit conversion**.

4 Some widening coercion is built directly into the VES operations on the built-in types (see [Section 11.1](#)). All
5 other coercion shall be explicitly requested. For the built-in types, the CTS provides operations to perform
6 widening coercions with no runtime checks and narrowing coercions with runtime checks.

7 7.3.3 Casting

8 Since a value can be of more than one type, a use of the value needs to clearly identify which of its types is
9 being used. Since values are read from locations that are typed, the type of the value which is used is the type
10 of the location from which the value was read. If a different type is to be used, the value is **cast** to one of its
11 other types. Casting is usually a compile time operation, but if the compiler cannot statically know that the
12 value is of the target type, a runtime cast check is done. Unlike coercion, a cast never changes the actual type of
13 an object nor does it change the representation. Casting preserves the identity of objects.

14 For example, a runtime check may be needed when casting a value read from a location that is typed as holding
15 values of a particular interface. Since an interface is an incomplete description of the value, casting that value
16 to be of a different interface type will usually result in a runtime cast check.

17 7.4 Type Members

18 As stated above, the type defines the allowable values and the allowable operations supported by the values of
19 the type. If the allowable values of the type have a substructure, that substructure is described via fields or array
20 elements of the type. If there are operations that are part of the type, those operations are described via methods
21 on the type. Fields, array elements, and methods are called **members** of the type. Properties and events are also
22 members of the type.

23 7.4.1 Fields, Array Elements, and Values

24 The representation of a value (except for those of built-in types) can be subdivided into sub-values. These sub-
25 values are either named, in which case they are called **fields**, or they are accessed by an indexing expression, in
26 which case they are called **array elements**. Types that describe values composed of array elements are **array**
27 types. Types that describe values composed of fields are **compound types**. A value cannot contain both fields
28 and array elements, although a field of a compound type may be an array type and an array element may be a
29 compound type.

30 Array elements and fields are typed, and these types never change. All of the array elements shall have the
31 same type. Each field of a compound type may have a different type.

32 7.4.2 Methods

33 A type may associate operations with the type or with each instance of the type. Such operations are called
34 **methods**. A method is named, and has a signature (see [clause 7.6.1](#)) that specifies the allowable types for all of
35 its arguments and for its return value, if any.

36 A method that is associated only with the type itself (as opposed to a particular instance of the type) is called a
37 **static method** (see [clause 7.4.3](#)).

38 A method that is associated with an instance of the type is either an **instance method** or a **virtual method** (see
39 [clause 7.4.4](#)). When they are invoked, instance and virtual methods are passed the instance on which this
40 invocation is to operate (known as **this** or a **this pointer**).

41 The fundamental difference between an instance method and a virtual method is in how the implementation is
42 located. An instance method is invoked by specifying a class and the instance method within that class. The
43 object passed as **this** may be **null** (a special value indicating that no instance is being specified) or an instance
44 of any type that inherits (see [clause 7.9.8](#)) from the class that defines the method. A virtual method may also be
45 called in this manner. This occurs, for example, when an implementation of a virtual method wishes to call the
46 implementation supplied by its parent class. The CTS allows **this** to be **null** inside the body of a virtual method.

1 **Rationale:** *Allowing a virtual method to be called with a non-virtual call eliminates the need for a "call super"*
2 **instruction and allows version changes between virtual and non-virtual methods. It requires CIL generators to**
3 **insert explicit tests for a null pointer if they don't want the null this pointer to propagate to called methods.**

4 A virtual or instance method may also be called by a different mechanism, a **virtual call**. Any type that inherits
5 from a type that defines a virtual method may provide its own implementation of that method (this is known as
6 **overriding**, see [clause 7.10.4](#)). It is the exact type of the object (determined at runtime) that is used to decide
7 which of the implementations to invoke

8 7.4.3 Static Fields and Static Methods

9 Types may declare locations that are associated with the type rather than any particular value of the type. Such
10 locations are **static fields** of the type. As such, static fields declare a location that is shared by all values of the
11 type. Just like non-static (instance) fields, a static field is typed and that type never changes. Static fields are
12 always restricted to a single application domain basis (see [Section 11.5](#)), but they may also be allocated on a
13 per-thread basis.

14 Similarly, types may also declare methods that are associated with the type rather than with values of the type.
15 Such methods are **static methods** of the type. Since an invocation of a static method does not have an
16 associated value on which the static method operates, there is no **this** pointer available within a static method.

17 7.4.4 Virtual Methods

18 An object type may declare any of its methods as **virtual**. Unlike other methods, each exact type that
19 implements the type may provide its own implementation of a virtual method. A virtual method may be
20 invoked through the ordinary method call mechanism that uses the static type, method name, and types of
21 parameters to choose an implementation, in which case the **this** pointer may be **null**. In addition, however, a
22 virtual method may be invoked by a special mechanism (a **virtual call**) that chooses the implementation based
23 on the dynamically detected type of the instance used to make the virtual call rather than the type statically
24 known at compile time. Virtual methods may be marked **final** (see [clause 7.10.2](#)).

25 7.5 Naming

26 Names are given to entities of the type system so that they can be referred to by other parts of the type system
27 or by the implementations of the types. Types, fields, methods, properties and events have names. With respect
28 to the type system values, locals, and parameters do not have names. An entity of the type system is given a
29 single name, e.g. there is only one name for a type.

30 7.5.1 Valid Names

31 All comparisons are done on a byte-by-byte (i.e. case sensitive, locale-independent, also known as code-point
32 comparison) basis. Where names are used to access built-in VES-supplied functionality (for example, the class
33 initialization method) there is always an accompanying indication on the definition so as not to build in any set
34 of reserved names.

35 **CLS Rule 4:** Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 (ISBN 0-
36 201-61633-5) governing the set of characters permitted to start and be included in identifiers, available on-line
37 at <http://www.unicode.org/unicode/reports/tr15/tr15-18.html>. Identifiers shall be in the canonical format
38 defined by Unicode Normalization Form C. For CLS purposes, two identifiers are the same if their lowercase
39 mappings (as specified by the Unicode locale-insensitive, 1-1 lowercase mappings) are the same. That is, for
40 two identifiers to be considered different under the CLS they shall differ in more than simply their case.
41 However, in order to override an inherited definition the CLI requires the precise encoding of the original
42 declaration be used.

43 **Note:**

44 **CLS (consumer):** need not consume types that violate CLS rule 4, but shall have a mechanism to allow access
45 to named items that use one of its own keywords as the name.

46 **CLS (extender):** need not create types that violate CLS rule 4. Shall provide a mechanism for defining new
47 names that obey these rules but are the same as a keyword in the language.

1 **CLS (framework):** shall not export types that violate CLS rule 4. Should avoid the use of names that are
2 commonly used as keywords in programming languages (see [Partition V Annex D](#))

3 **7.5.2 Assemblies and Scoping**

4 Generally, names are not unique. Names are collected into groupings called **scopes**. Within a scope, a name
5 may refer to multiple entities as long as they are of different kinds (methods, fields, nested types, properties,
6 and events) or have different signatures.

7 **CLS Rule 5:** All names introduced in a CLS-compliant scope shall be distinct independent of kind, except
8 where the names are identical and resolved via overloading. That is, while the CTS allows a single type to use
9 the same name for a method and a field, the CLS does not.

10 **CLS Rule 6:** Fields and nested types shall be distinct by identifier comparison alone, even though the CTS
11 allows distinct signatures to be distinguished. Methods, properties, and events that have the same name (by
12 identifier comparison) shall differ by more than just the return type, except as specified in CLS Rule 39.

13 **Note:**

14 **CLS (consumer):** need not consume types that violate these rules after ignoring any members that are marked
15 as not CLS-compliant.

16 **CLS (extender):** need not provide syntax for defining types that violate these rules.

17 **CLS (framework):** shall not mark types as CLS-compliant if they violate these rules unless they mark
18 sufficient offending items within the type as not CLS-compliant so that the remaining members do not conflict
19 with one another.

20 A named entity has its name in exactly one scope. Hence, to identify a named entity, both a scope and a name
21 need to be supplied. The scope is said to **qualify** the name. Types provide a scope for the names in the type;
22 hence types qualify the names in the type. For example, consider a compound type **Point** that has a field
23 named **x**. The name “field **x**” by itself does not uniquely identify the named field, but the **qualified name**
24 “field **x** in type **Point**” does.

25 Since types are named, the names of types are also grouped into scopes. To fully identify a type, the type name
26 shall be qualified by the scope that includes the type name. Type names are scoped by the **assembly** that
27 contains the implementation of the type. An assembly is a configured set of loadable code modules and other
28 resources that together implement a unit of functionality. The type name is said to be in the **assembly scope** of
29 the assembly that implements the type. Assemblies themselves have names that form the basis of the
30 CTS naming hierarchy.

31 The **type definition**:

- 32 • Defines a name for the type being defined, i.e. the **type name**, and specifies a scope in which that
33 name will be found
- 34 • Defines a **member scope** in which the names of the different kinds of members (fields, methods,
35 events, and properties) are bound. The tuple of (member name, member kind, and member
36 signature) is unique within a member scope of a type.
- 37 • Implicitly assigns the type to the assembly scope of the assembly that contains the type definition.

38 The CTS supports an **enum** (also known as an **enumeration type**), an alternate name for an existing type. For
39 purposes of matching signatures an enum shall not be the same as the underlying type. Instances of an enum,
40 however, shall be assignment compatible with the underlying type and vice versa. That is: no cast (see
41 clause 7.3.3) or coercion (see clause 7.3.2) is required to convert from the enum to the underlying type, nor are
42 they required from the underlying type to the enum. An enum is considerably more restricted than a true type:

- 43 • It shall have exactly one instance field, and the type of that field defines the underlying type of
44 the enumeration.
- 45 • It shall not have any methods of its own.
- 46 • It shall derive from **System.Enum** (see [Partition IV](#)).

- 1 • It shall not implement any interfaces of its own.
- 2 • It shall not have any properties or events of its own.
- 3 • It shall not have any static fields unless they are literal (see [clause 7.6.1](#)).

4 The underlying type shall be a built-in integer type. Enums shall derive from `System.Enum`, hence they are
5 value types. Like all value types, they shall be sealed (see [clause 7.9.9](#)).

6 **CLS Rule 7:** The underlying type of an enum shall be a built-in CLS integer type.

7 **CLS Rule 8:** There are two distinct kinds of enums, indicated by the presence or absence of the
8 `System.FlagsAttribute` (see [Partition IV](#)) custom attribute. One represents named integer values, the other
9 represents named bit flags that can be combined to generate an unnamed value. The value of an enum is not limited to the
10 specified values.

11 **CLS Rule 9:** Literal static fields (see [clause 7.6.1](#)) of an enum shall have the type of the enum itself.

12 **Note:**

13 **CLS (consumer):** Shall accept definition of enums that follow these rules, but need not distinguish flags from
14 named values.

15 **CLS (extender):** Same as consumer. Extender languages are encouraged to allow the authoring of enums, but
16 need not do so.

17 **CLS (framework):** shall not expose enums that violate these rules; and shall not assume that enums have only
18 the specified values (even for enums that are named values).

19 7.5.3 Visibility, Accessibility, and Security

20 To refer to a named entity in a scope, both the scope and the name in the scope shall be visible (see
21 [clause 7.5.3.1](#)). Visibility is determined by the relationship between the entity that contains the reference (the
22 referent) and the entity that contains the name being referenced. Consider the following pseudo-code:

```
23    class A
24    { int32 IntInsideA;
25    }
26    class B inherits from A
27    { method X(int32, int32) returning Boolean
28    { IntInsideA := 15;
29    }
30 }
```

31 If we consider the reference to the field `IntInsideA` in class A:

- 32 • We call class B the **referent** because it has a method that refers to that field.
- 33 • We call `IntInsideA` in class A the **referenced entity**.

34 There are two fundamental questions that need to be answered in order to decide whether the referent is
35 allowed to access the referenced entity. The first is whether the name of the referenced entity is **visible** to the
36 referent. If it is visible, then there is a separate question of whether the referent is **accessible** (see
37 [clause 7.5.3.2](#)).

38 Access to a member of a type is permitted only if all three of the following conditions are met:

- 39 1. The type is visible.
- 40 2. The member is accessible.
- 41 3. All relevant security demands (see [clause 7.5.3.3](#)) have been granted.

42 7.5.3.1 Visibility of Types

43 Only type names, not member names, have controlled visibility. Type names fall into one of the following three
44 categories

- 1 • **Exported** from the assembly in which they are defined. While a type may be marked to *allow* it
2 to be exported from the assembly, it is the configuration of the assembly that decides whether the
3 type name *is* made available.
- 4 • **Not exported** outside the assembly in which they are defined.
- 5 • Nested within another type. In this case, the type itself has the visibility of the type inside of
6 which it is nested (its **enclosing type**). See clause 7.5.3.4.

7 7.5.3.2 Accessibility of Members

8 A type scopes all of its members, and it also specifies the accessibility rules for its members. Except where
9 noted, accessibility is decided based only on the statically visible type of the member being referenced and the
10 type and assembly that is making the reference. The CTS supports seven different rules for accessibility:

- 11 • **Compiler-Controlled** – accessible only through use of a definition, not a reference, hence only
12 accessible from within a single compilation unit and under the control of the compiler.
- 13 • **Private** – accessible only to referents in the implementation of the exact type that defines the
14 member.
- 15 • **Family** – accessible to referents that support the same type, i.e. an exact type and all of the types
16 that inherit from it. For verifiable code (see Section 7.8), there is an additional requirement that
17 may require a runtime check: the reference shall be made through an item whose exact type
18 supports the exact type of the referent. That is, the item whose member is being accessed shall
19 inherit from the type performing the access.
- 20 • **Assembly** – accessible only to referents in the same assembly that contains the implementation of
21 the type.
- 22 • **Family-and-Assembly** – accessible only to referents that qualify for both Family and Assembly
23 access.
- 24 • **Family-or-Assembly** – accessible only to referents that qualify for either Family or Assembly
25 access.
- 26 • **Public** – accessible to all referents.

27 In general, a member of a type can have any one of these accessibility rules assigned to it. There are two
28 exceptions, however:

- 29 1. Members defined by an interface shall be public.
- 30 2. When a type defines a virtual method that overrides an inherited definition, the accessibility shall
31 either be identical in the two definitions or the overriding definition shall permit more access than
32 the original definition. For example, it is possible to override an **assembly virtual** method with a
33 new implementation that is **public virtual**, but not with one that is **family virtual**. In the case of
34 overriding a definition derived from another assembly, it is not considered restricting access if the
35 base definition has **Family-or-Assembly** access and the override has only **family** access.

36 **Rationale:** *Languages including C++ allow this “widening” of access. Restricting access would provide an
37 incorrect illusion of security since simply casting an object to the base class (which occurs implicitly on
38 method call) would allow the method to be called despite the restricted accessibility. To prevent overriding a
39 virtual method use **final** (see clause 7.10.2) rather than relying on limited accessibility.*

40 **CLS Rule 10:** Accessibility shall not be changed when overriding inherited methods, except when overriding a
41 method inherited from a different assembly with accessibility **Family-or-Assembly**. In this case the override
42 shall have accessibility **family**.

43 **Note:**

44 **CLS (consumer):** need not accept types that widen access to inherited virtual methods.

45 **CLS (extender):** need not provide syntax to widen access to inherited virtual methods.

1 CLS (frameworks) shall not rely on the ability to widen access to a virtual method, either in the exposed
2 portion of the framework or by users of the framework.

3 7.5.3.3 Security Permissions

4 Access to members is also controlled by security demands that may be attached to an assembly, type, method,
5 property, or event. Security demands are not part of a type contract (see [Section 7.6](#)), and hence are not
6 inherited. There are two kinds of demands:

- 7 • An **inheritance demand**. When attached to a type it requires that any type that wishes to inherit
8 from this type shall have the specified security permission. When attached to a non-final virtual
9 method it requires that any type that wishes to override this method shall have the specified
10 permission. It shall not be attached to any other member.
- 11 • A **reference demand**. Any attempt to resolve a reference to the marked item shall have specified
12 security permission.

13 Only one demand of each kind may be attached to any item. Attaching a security demand to an assembly
14 implies that it is attached to all types in the assembly unless another demand of the same kind is attached to the
15 type. Similarly, a demand attached to a type implies the same demand for all members of the type unless
16 another demand of the same kind is attached to the member. For additional information, see Declarative
17 Security in [Partition II](#), and the classes in the `System.Security` namespace in [Partition IV](#).

18 7.5.3.4 Nested Types

19 A type (called a nested type) can be a member of an enclosing type. A nested type has the same visibility as the
20 enclosing type and has an accessibility as would any other member of the enclosing type. This accessibility
21 determines which other types may make references to the nested type. That is, for a class to define a field or
22 array element of a nested type, have a method that takes a nested type as a parameter or returns one as value,
23 etc., the nested type shall be both visible and accessible to the referencing type. A nested type is part of the
24 enclosing type so its methods have access to all members of its enclosing type, as well as family access to
25 members of the type from which it inherits (see [clause 7.9.8](#)). The names of nested types are scoped by their
26 enclosing type, not their assembly (only top-level types are scoped by their assembly). There is no requirement
27 that the names of nested types be unique within an assembly.

28 7.6 Contracts

29 Contracts are named. They are the shared assumptions on a set of signatures (see [clause 7.6.1](#)) between all
30 implementers and all users of the contract. The signatures are the part of the contract that can be checked and
31 enforced.

32 Contracts are not types; rather they specify requirements on the implementation of types. Types state which
33 contracts they abide by, i.e. which contracts all implementations of the type shall support. An implementation
34 of a type can be verified to check that the enforceable parts of a contract, the named signatures, have been
35 implemented. The kinds of contracts are:

- 36 • **Class contract** – A class contract is specified with a class definition. Hence, a class definition
37 defines both the class contract and the **class type**. The name of the class contract and the name of
38 the class type are the same. A class contract specifies the representation of the values of the class
39 type. Additionally, a class contract specifies the other contracts that the class type supports, e.g.,
40 which interfaces, methods, properties and events shall be implemented. A class contract, and
41 hence the class type, can be supported by other class types as well. A class type that supports the
42 class contract of another class type is said to **inherit** from that class type.
- 43 • **Interface contract** – An interface contract is specified with an interface definition. Hence, an
44 interface definition defines both the interface contract and the **interface type**. The name of the
45 interface contract and the name of the interface type are the same. Many types can support an
46 interface contract. Like a class contract, interface contracts specify which other contracts the
47 interface supports, e.g. which interfaces, methods, properties and events shall be implemented.

1 **Note:** An interface type can never fully describe the representation of a value. Therefore an interface
2 type can never support a class contract, and hence can never be a class type or an exact type.

3 • **Method contract** – A method contract is specified with a method definition. A method contract
4 is a named operation that specifies the contract between the implementation(s) of the method and
5 the callers of the method. A method contract is always part of a type contract (class, value type,
6 or interface), and describes how a particular named operation is implemented. The method
7 contract specifies the contracts that each parameter to the method shall support and the contracts
8 that the return value shall support, if there is a return value.

9 • **Property contract** – A property contract is specified with a property definition. There is an
10 extensible set of operations for handling a named value, which includes a standard pair for
11 reading the value and changing the value. A property contract specifies method contracts for the
12 subset of these operations that shall be implemented by any type that supports the property
13 contract. A type can support many property contracts, but any given property contract can be
14 supported by exactly one type. Hence, property definitions are a part of the type definition of the
15 type that supports the property.

16 • **Event contract** – An event contract is specified with an event definition. There is an extensible
17 set of operations for managing a named event, which includes three standard methods (register
18 interest in an event, revoke interest in an event, fire the event). An event contract specifies
19 method contracts for all of the operations that shall be implemented by any type that supports the
20 event contract. A type can support many event contracts, but any given event contract can be
21 supported by exactly one type. Hence, event definitions are a part of the type definition of the
22 type that supports the event.

23 **7.6.1 Signatures**

24 Signatures are the part of a contract that can be checked and automatically enforced. Signatures are formed by
25 adding constraints to types and other signatures. A constraint is a limitation on the use of or allowed operations
26 on a value or location. Example constraints would be whether a location may be overwritten with a different
27 value or whether a value may ever be changed.

28 All locations have signatures, as do all values. Assignment compatibility requires that the signature of the
29 value, including constraints, is compatible with the signature of the location, including constraints. There are
30 four fundamental kinds of signatures: type signatures, location signatures, parameter signatures, and method
31 signatures.

32 **CLS Rule 11:** All types appearing in a signature shall be CLS-compliant.

33 **CLS Rule 12:** The visibility and accessibility of types and members shall be such that types in the signature of
34 any member shall be visible and accessible whenever the member itself is visible and accessible. For example,
35 a public method that is visible outside its assembly shall not have an argument whose type is visible only
36 within the assembly.

37 **Note:**

38 **CLS (consumer):** need not accept types whose members violate these rules.

39 **CLS (extender):** need not provide syntax to violate these rules.

40 **CLS (framework):** shall not violate this rule in its exposed types and their members.

41 The following sections describe the various kinds of signatures. These descriptions are cumulative: the simplest
42 signature is a type signature; a location signature is a type signature plus (optionally) some additional attributes;
43 and so forth.

44 **7.6.1.1 Type Signatures**

45 Type signatures define the constraints on a value and its usage. A type, by itself, is a valid type signature. The
46 type signature of a value cannot be determined by examining the value or even by knowing the class type of the
47 value. The type signature of a value is derived from the location signature (see below) of the location from

1 which the value is loaded. Normally the type signature of a value is the type in the location signature from
2 which the value is loaded.

3 **Rationale:** *The distinction between a Type Signature and a Location Signature (below) is not currently useful.*
4 *It is made because certain constraints, such as "constant," are constraints on values not locations. Future*
5 *versions of this standard, or non-standard extensions, may introduce type constraints, thus making the*
6 *distinction meaningful.*

7 7.6.1.2 **Location Signatures**

8 All locations are typed. This means that all locations have a **location signature**, which defines constraints on
9 the location, its usage, and on the usage of the values stored in the location. Any valid type signature is a valid
10 location signature. Hence, a location signature contains a type and may additionally contain the **constant**
11 constraint. The location signature may also contain **location constraints** that give further restrictions on the
12 uses of the location. The location constraints are:

- 13 • The **init-only constraint** promises (hence, requires) that once the location has been initialized, its
14 contents never change. Namely, the contents are initialized before any access, and after
15 initialization, no value may be stored in the location. The contents are always identical to the
16 initialized value (see [clause 7.2.3](#)). This constraint, while logically applicable to any location,
17 shall only be placed on fields (static or instance) of compound types.
- 18 • The **literal constraint** promises that the value of the location is actually a fixed value of a built-in
19 type. The value is specified as part of the constraint. Compilers are required to replace all
20 references to the location with its value, and the VES therefore need not allocate space for the
21 location. This constraint, while logically applicable to any location, shall only be placed on static
22 fields of compound types. Fields that are so marked are not permitted to be referenced from CIL.
23 (they shall be in-lined to their constant value at compile time), but are available using Reflection
24 and tools that directly deal with the metadata.

25 **CLS Rule 13:** The value of a literal static is specified through the use of field initialization metadata (see
26 [Partition II](#)). A CLS compliant literal must have a value specified in field initialization metadata that is of
27 exactly the same type as the literal (or of the underlying type, if that literal is an **enum**).

28 **Note:**

29 **CLS (consumer):** must be able to read field initialization metadata for static literal fields and inline the value
30 specified when referenced. Consumers may assume that the type of the field initialization metadata is exactly
31 the same as the type of the literal field, i.e., a consumer tool need not implement conversions of the values.

32 **CLS (extender):** must avoid producing field initialization metadata for static literal fields in which the type of
33 the field initialization metadata does not exactly match the type of the field.

34 **CLS (framework):** should avoid the use of syntax specifying a value of a literal that requires conversion of the
35 value. Note that compilers may do the conversion themselves before persisting the field initialization metadata
36 resulting in a CLS compliant framework, but frameworks are encouraged not to rely on such implicit
37 conversions.

38 **Note:** It might seem reasonable to provide a **volatile** constraint on a location that would require that the value
39 stored in the location not be cached between accesses. Instead, CIL includes a **volatile** prefix to certain
40 instructions to specify that the value neither be cached nor computed using an existing cache. Such a constraint
41 may be encoded using a custom attribute (see [Section 8.7](#)), although this standard does not specify such an
42 attribute.

44 7.6.1.3 **Local Signatures**

45 A **local signature** specifies the contract on a local variable allocated during the running of a method. A local
46 signature contains a full location signature, plus it may specify one additional constraint:

47 The **byref** constraint states that the content of the corresponding location is a **managed pointer**. A managed
48 pointer may point to a local variable, parameter, field of a compound type, or element of an array. However,

1 when a call crosses a remoting boundary (see [Section 11.5](#)) a conforming implementation may use a copy-in/copy-out mechanism instead of a managed pointer. Thus programs shall not rely on the aliasing behavior of
2 true pointers.
3

4 In addition, there is one special local signature. The **typed reference** local variable signature states that the
5 local will contain both a managed pointer to a location and a runtime representation of the type that may be
6 stored at that location. A typed reference signature is similar to a **byref** constraint, but while the **byref** specifies
7 the type as part of the **byref** constraint (and hence as part of the type description), a typed reference provides the
8 type information dynamically. A typed reference is a full signature in itself and can not be combined with other
9 constraints. In particular, it is not possible to specify a **byref** whose type is **typed reference**.

10 The typed reference signature is actually represented as a built-in value type, like the integer and floating point
11 types. In the Base Class Library (see [Partition IV](#)) the type is known as **System.TypedReference** and in the
12 assembly language used in [Partition II](#) it is designated by the keyword **typedref**. This type shall only be used
13 for parameters and local variables. It shall not be boxed, nor shall it be used as the type of a field, element of an
14 array, return value, etc.

15 **CLS Rule 14:** Typed references are not CLS-compliant.

16 **Note:**

17 **CLS (consumer):** there is no need to accept this type.

18 **CLS (extender):** there is no need to provide syntax to define this type or to extend interfaces or classes that use
19 this type.

20 **CLS (framework):** this type shall not appear in exposed members.

21 7.6.1.4 Parameter Signatures

22 **Parameter signatures** define constraints on how an individual value is passed as part of a method invocation.
23 Parameter signatures are declared by method definitions. Any valid local signature is a valid parameter
24 signature.

25 7.6.1.5 Method Signatures

26 **Method signatures** are composed of:

- 27 • a calling convention,
- 28 • a list of zero or more parameter signatures, one for each parameter of the method,
- 29 • and a type signature for the result value if one is produced.

30 Method signatures are declared by method definitions. Only one constraint can be added to a method signature
31 in addition to those of parameter signatures:

- 32 • The **varargs** constraint may be included to indicate that all arguments past this point are optional.
33 When it appears, the calling convention shall be one that supports variable argument lists.

34 Method signatures are used in two different ways. They are used as part of a method definition and as a
35 description of a calling site when calling through a function pointer. In this latter case, the method signature
36 indicates

- 37 • the calling convention (which may include platform-specific calling conventions)
- 38 • the type of all the argument values that are being passed,
- 39 • if needed, a varargs marker indicating where the fixed parameter list ends and the variable
40 parameter list begins

41 When used as part of a method definition, the varargs constraint is represented by the choice of calling
42 convention.

43 **CLS Rule 15:** The varargs constraint is not part of the CLS, and the only calling convention supported by the
44 CLS is the standard managed calling convention.

1 **Note:**

2 **CLS (consumer):** there is no need to accept methods with variable argument lists or unmanaged calling
3 conventions.

4 **CLS (extender):** there is no need to provide syntax to declare varargs methods or unmanaged calling
5 conventions.

6 **CLS (framework):** neither varargs methods nor methods with unmanaged calling conventions may be exposed
7 externally.

8 **7.7 Assignment Compatibility**

9 The constraints in the type signature and the location signature affect assignment compatibility of a value to a
10 location. Assignment compatibility of a value (described by a type signature) to a location (described by a
11 location signature) is defined as follows:

12 One of the types supported by the exact type of the value is the same as the type in the location signature.

13 This allows, for example, an instance of a class that inherits from a base class (hence supports the base class's
14 type contract) to be stored into a location whose type is that of the base class.

15 **7.8 Type Safety and Verification**

16 Since types specify contracts, it is important to know whether a given implementation lives up to these
17 contracts. An implementation that lives up to the enforceable part of the contract (the named signatures) is said
18 to be **typesafe**. An important part of the contract deals with restrictions on the visibility and accessibility of
19 named items as well as the mapping of names to implementations and locations in memory.

20 Typesafe implementations only store values described by a type signature in a location that is assignment
21 compatible with the location signature of the location (see [clause 7.6.1](#)). Typesafe implementations never apply
22 an operation to a value that is not defined by the exact type of the value. Typesafe implementations only access
23 locations that are both visible and accessible to them. In a typesafe implementation, the exact type of a value
24 cannot change.

25 **Verification** is a mechanical process of examining an implementation and asserting that it is typesafe.
26 Verification is said to succeed if the process proves that an implementation is typesafe. Verification is said to
27 fail if that process does not prove the type safety of an implementation. Verification is necessarily conservative:
28 it may report failure for a typesafe implementation, but it never reports success for an implementation that is
29 not typesafe. For example, most verification processes report implementations that do pointer-based arithmetic
30 as failing verification, even if the implementation is in fact typesafe.

31 There are many different processes that can be the basis of verification. The simplest possible process simply
32 says that all implementations are not typesafe. While correct and efficient this is clearly not particularly useful.
33 By spending more resources (time and space) a process can correctly identify more typesafe implementations.
34 It has been proven, however, that no mechanical process can in finite time and with no errors correctly identify
35 all implementations as either typesafe or not typesafe. The choice of a particular verification process is thus a
36 matter of engineering, based on the resources available to make the decision and the importance of detecting
37 the typesafety of different programming constructs.

38 **7.9 Type Definers**

39 Type definers construct a new type from existing types. **Implicit types** (e.g., built-in types, arrays, and pointers
40 including function pointers) are defined when they are used. The mention of an implicit type in a signature is in
41 and of itself a complete definition of the type. Implicit types allow the VES to manufacture instances with a
42 standard set of members, interfaces, etc. Implicit types need not have user-supplied names.

43 All other types shall be explicitly defined using an explicit type definition. The explicit type definers are:

- 44 • interface definitions – used to define interface types
- 45 • class definitions – used to define:

- 1 o object types
- 2 o value types and their associated boxed types

3 **Note:** While class definitions always define class types, not all class types require a class definition. Array
4 types and pointer types, which are implicitly defined, are also class types. See [clause 7.2.3](#).

5 Similarly, not all types defined by a class definition are object types. Array types, explicitly defined object
6 types, and boxed types are object types. Pointer types, function pointer types, and value types are not object
7 types. See [clause 7.2.3](#).

8 7.9.1 Array Types

9 An **array type** shall be defined by specifying the element type of the array, the **rank** (number of dimensions)
10 of the array, and the upper and lower bounds of each dimension of the array. Hence, no separate definition of
11 the array type is needed. The bounds (as well as indices into the array) shall be signed integers. While the
12 actual bounds for each dimension are known at runtime, the signature may specify the information that is
13 known at compile time: no bounds, a lower bound, or both an upper and lower bound.

14 Array elements shall be laid out within the array object in row-major order, i.e. the elements associated with the
15 rightmost array dimension shall be laid out contiguously from lowest to highest index. The actual storage
16 allocated for each array element may include platform-specific padding.

17 Values of an array type are objects; hence an array type is a kind of object type (see [clause 7.2.3](#)). Array objects
18 are defined by the CTS to be a repetition of locations where values of the array element type are stored. The
19 number of repeated values is determined by the rank and bounds of the array.

20 Only type signatures, not location signatures, are allowed as array element types.

21 Exact array types are created automatically by the VES when they are required. Hence, the operations on an
22 array type are defined by the CTS. These generally are: allocating the array based on size and lower bound
23 information, indexing the array to read and write a value, computing the address of an element of the array (a
24 managed pointer), and querying for the rank, bounds, and the total number of values stored in the array.

25 **CLS Rule 16:** Arrays shall have elements with a CLS-compliant type and all dimensions of the array shall
26 have lower bounds of zero. Only the fact that an item is an array and the element type of the array shall be
27 required to distinguish between overloads. When overloading is based on two or more array types the element
28 types shall be named types.

29 **Note:** so-called "jagged arrays" are CLS-compliant, but when overloading multiple array types they are one-
30 dimensional, zero-based arrays of type `System.Array`.

31 **CLS (consumer):** there is no need to support arrays of non-CLS types, even when dealing with instances of
32 `System.Array`. Overload resolution need not be aware of the full complexity of array types. Programmers
33 should have access to the Get, Set, and Address methods on instances of `System.Array` if there is no language
34 syntax for the full range of array types.

35 **CLS (extender):** there is no need to provide syntax to define non-CLS types of arrays or to extend interfaces or
36 classes that use non-CLS array types. Shall provide access to the type `System.Array`, but may assume that all
37 instances will have a CLS-compliant type. While the full array signature must be used to override an inherited
38 method that has an array parameter, the full complexity of array types need not be made visible to
39 programmers. Programmers should have access to the Get, Set, and Address methods on instances of
40 `System.Array` if there is no language syntax for the full range of array types.

41 **CLS (framework):** non-CLS array types shall not appear in exposed members. Where possible, use only one-
42 dimensional, zero-based arrays (vectors) of simple named types, since these are supported in the widest range
43 of programming languages. Overloading on array types should be avoided, and when used shall obey the
44 restrictions.

45 Array types form a hierarchy, with all array types inheriting from the type `System.Array`. This is an abstract
46 class (see [clause 7.9.6.2](#)) that represents all arrays regardless of the type of their elements, their rank, or their
47 upper and lower bounds. The VES creates one array type for each distinguishable array type. In general, array
48 types are only distinguished by the type of their elements and their rank. The VES, however, treats single

1 dimensional, zero-based arrays (also known as **vectors**) specially. Vectors are also distinguished by the type of
2 their elements, but a vector is distinct from a single-dimensional array of the same element type that has a non-
3 zero lower bound. Zero-dimensional arrays are not supported.

4 Consider the following examples, using the syntax of CIL as described in Partition II:

5

1

Table 2: Array Examples

Static specification of type	Actual type constructed	Allowed in CLS?
int32[]	vector of int32	Yes
int32[0..5]	vector of int32	Yes
int32[1..5]	array, rank 1, of int32	No
int32[,]	array, rank 2, of int32	Yes
int32[0..3, 0..5]	array, rank 2, of int32	Yes
int32[0.., 0..]	array, rank 2, of int32	Yes
int32[1.., 0..]	array, rank 2, of int32	No

2

3 7.9.2 Unmanaged Pointer Types

4 An **unmanaged pointer type** (also known simply as a “pointer type”) is defined by specifying a location
5 signature for the location the pointer references. Any signature of a pointer type includes this location
6 signature. Hence, no separate definition of the pointer type is needed.

7 While pointer types are Reference Types, values of a pointer type are not objects (see [clause 7.2.3](#)), and hence
8 it is not possible, given a value of a pointer type, to determine its exact type. The CTS provides two typesafe
9 operations on pointer types: one to load the value from the location referenced by the pointer and the other to
10 store an assignment compatible value into that location. The CTS also provides three operations on pointer
11 types (byte-based address arithmetic): adding and subtracting integers from pointers, and subtracting one
12 pointer from another. The results of the first two operations are pointers to the same type signature as the
13 original pointer. See [Partition III](#) for details.

14 **CLS Rule 17:** Unmanaged pointer types are not CLS-compliant.

15 **Note:**

16 **CLS (consumer):** there is no need to support unmanaged pointer types.

17 **CLS (extender):** there is no need to provide syntax to define or access unmanaged pointer types.

18 **CLS (framework):** unmanaged pointer types shall not be externally exposed.

19 7.9.3 Delegates

20 Delegates are the object-oriented equivalent of function pointers. Unlike function pointers, delegates are
21 object-oriented, type-safe, and secure. Delegates are created by defining a class that derives from the base type
22 [System.Delegate](#) (see [Partition IV](#)). Each delegate type shall provide a method named **Invoke** with appropriate
23 parameters, and each instance of a delegate forwards calls to its **Invoke** method to a compatible static or
24 instance method on a particular object. The object and method to which it delegates are chosen when the
25 delegate instance is created.

26 In addition to an instance constructor and an **Invoke** method, delegates may optionally have two additional
27 methods: **BeginInvoke** and **EndInvoke**. These are used for asynchronous calls.

28 While, for the most part, delegates appear to be simply another kind of user defined class, they are tightly
29 controlled. The implementations of the methods are provided by the VES, not user code. The only additional
30 members that may be defined on delegate types are static or instance methods.

31 7.9.4 Interface Type Definition

32 An **interface definition** defines an interface type. An interface type is a named group of methods, locations and
33 other contracts that shall be implemented by any object type that supports the interface contract of the same
34 name. An interface definition is always an incomplete description of a value, and as such can never define a
35 class type or an exact type, nor can it be an object type.

1 Zero or more object types can support an interface type, and only object types can support an interface type. An
2 interface type may require that objects that support it shall also support other (specified) interface types. An
3 object type that supports the named interface contract shall provide a complete implementation of the methods,
4 locations, and other contracts specified (but not implemented by) the interface type. Hence, a value of an object
5 type is also a value of all of the interface types the object type supports. Support for an interface contract is
6 declared, never inferred, i.e. the existence of implementations of the methods, locations, and other contracts
7 required by the interface type does not imply support of the interface contract.

8 **CLS Rule 18:** CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in
9 order to implement them.

10 **Note:**

11 **CLS (consumer):** there is no need to deal with such interfaces.

12 **CLS (extender):** need not provide a mechanism for defining such interfaces.

13 **CLS (framework):** shall not expose any non-CLS compliant methods on interfaces it defines for external use.

14 Interfaces types are necessarily incomplete since they say nothing about the representation of the values of the
15 interface type. For this reason, an interface type definition shall not provide field definitions for values of the
16 interface type (i.e. instance fields), although it may declare static fields (see [clause 7.4.3](#)).

17 Similarly, an interface type definition shall not provide implementations for any methods on the values of its
18 type. However, an interface type definition may and usually does define method contracts (method name and
19 method signature) that shall be implemented by supporting types. An interface type definition may define and
20 implement static methods (see [clause 7.4.3](#)) since static methods are associated with the interface type itself
21 rather than with any value of the type.

22 Interfaces may have static or virtual methods, but shall not have instance methods.

23 **CLS Rule 19:** CLS-compliant interfaces shall not define static methods, nor shall they define fields.

24 **Note:**

25 **CLS-compliant interfaces** may define properties, events, and virtual methods.

26 **CLS (consumer):** need not accept interfaces that violate these rules.

27 **CLS (extender):** need not provide syntax to author interfaces that violate these rules.

28 **CLS (framework):** shall not externally expose interfaces that violate these rules. Where static methods,
29 instance methods, or fields are required a separate class may be defined that provides them.

30 Interface types may also define event and property contracts that shall be implemented by object types that
31 support the interface. Since event and property contracts reduce to sets of method contracts ([Section 7.6](#)), the
32 above rules for method definitions apply. For more information, see [clause 7.11.4](#) and [clause 7.11.3](#).

33 Interface type definitions may specify other interface contracts that implementations of the interface type are
34 required to support. See [clause 7.9.11](#) for specifics.

35 An interface type is given a visibility attribute, as described in [clause 7.5.3](#), that controls from where the
36 interface type may be referenced. An interface type definition is separate from any object type definition that
37 supports the interface type. Hence, it is possible, and often desirable, to have a different visibility for the
38 interface type and the implementing object type. However, since accessibility attributes are relative to the
39 implementing type rather than the interface itself, all members of an interface shall have public accessibility,
40 and no security permissions may be attached to members or to the interface itself.

41 7.9.5 Class Type Definition

42 All types other than interfaces, and those types for which a definition is automatically supplied by the CTS, are
43 defined by class definitions. A class type is a complete specification of the representation of the values of the
44 class type and all of the contracts (class, interface, method, property, and event) that are supported by the class
45 type. Hence, a class type is an exact type. A class definition, unless it specifies that the class is an abstract

1 **object type**, not only defines the class type: it also provides implementations for all of the contracts supported
2 by the class type.

3 A class definition, and hence the implementation of the class type, always resides in some assembly. An
4 assembly is a configured set of loadable code modules and other resources that together implement a unit of
5 functionality.

6 **Note:** While class definitions always define class types, not all class types require a class definition. Array
7 types and pointer types, which are implicitly defined, are also class types. See [clause 7.2.3](#).

8 An explicit class definition is used to define:

- An object type (see [clause 7.2.3](#)).
- A value type and its associated boxed type (see [clause 7.2.4](#)).

11 An explicit class definition:

- Names the class type.
- Implicitly assigns the class type name to a scope, i.e. the assembly that contains the class definition, (see [clause 7.5.2](#)).
- Defines the class contract of the same name (see [Section 7.6](#)).
- Defines the representations and valid operations of all values of the class type using member definitions for the fields, methods, properties, and events (see [Section 7.11](#)).
- Defines the static members of the class type (see [Section 7.11](#)).
- Specifies any other interface and class contracts also supported by the class type.
- Supplies implementations for member and interface contracts supported by the class type.
- Explicitly declares a visibility for the type, either public or assembly (see [clause 7.5.3](#)).
- May optionally specify a method to be called to initialize the type.

23 The semantics of when, and what triggers execution of such type initialization methods, is as follows:

1. A type may have a type-initializer method, or not.
2. A type may be specified as having a relaxed semantic for its type-initializer method (for convenience below, we call this relaxed semantic **BeforeFieldInit**)
3. If marked **BeforeFieldInit** then the type's initializer method is executed at, or sometime before, first access to any static field defined for that type
4. If *not* marked **BeforeFieldInit** then that type's initializer method is executed at (i.e., is triggered by):
 - o first access to any static or instance field of that type, or
 - o first invocation of any static, instance or virtual method of that type
5. Execution of any type's initializer method will *not* trigger automatic execution of any initializer methods defined by its base type, nor of any interfaces that the type implements

35 **Note:** **BeforeFieldInit** behavior is intended for initialization code with no interesting side-effects, where exact
36 timing does not matter. Also, under **BeforeFieldInit** semantics, type initializers are allowed to be executed *at*
37 *or before* first access to any static field of that Type --- at the discretion of the CLI.

38 If a language wishes to provide more rigid behavior -- e.g. type initialization automatically triggers execution
39 of parents initializers, in a top-to-bottom order, then it can do so by either:

- defining hidden static fields and code in each class constructor that touches the hidden static field of its parent and/or interfaces it implements; or

1 by making explicit calls to `System.Runtime.CompilerServices.RuntimeHelpers.RunClassConstructor` (see [Partition IV](#)).
2

3 **7.9.6 Object Type Definitions**

4 All objects are instances of an **object type**. The object type of an object is set when the object is created and it
5 is immutable. The object type describes the physical structure of the instance and the operations that are
6 allowed on it. All instances of the same object type have the same structure and the same allowable operations.
7 Object types are explicitly declared by a class type definition, with the exception of **Array** types, which are
8 intrinsically provided by the VES.

9 **7.9.6.1 Scope and Visibility**

10 Since object type definitions are class type definitions, object type definitions implicitly specify the scope of
11 the name of object type to be the assembly that contains the object type definition, see [clause 7.5.2](#). Similarly,
12 object type definitions shall also explicitly state the visibility attribute of the object type (either **public** or
13 **assembly**); see [clause 7.5.3](#).

14 **7.9.6.2 Concreteness**

15 An object type may be marked as **abstract** by the object type definition. An object type that is not marked
16 **abstract** is by definition **concrete**. Only object types may be declared as **abstract**. Only an **abstract** object type
17 is allowed to define method contracts for which the type or the VES does not also provide the implementation.
18 Such method contracts are called **abstract methods** (see [Section 7.11](#)). All methods on an **abstract class** need not
19 be **abstract**.

20 It is an error to attempt to create an instance of an **abstract object type**, whether or not the type has **abstract**
21 methods. An object type that derives from an **abstract object type** may be **concrete** if it provides
22 implementations for any **abstract methods** in the base object type and is not itself marked as **abstract**. Instances
23 may be made of such a **concrete derived class**. Locations may have an **abstract type**, and instances of a **concrete**
24 type that derives from the **abstract type** may be stored in them.

25 **7.9.6.3 Type Members**

26 Object type definitions include member definitions for all of the members of the type. Briefly, members of a
27 type include fields into which values are stored, methods that may be invoked, properties that are available, and
28 events that may be raised. Each member of a type may have attributes as described in [Section 7.4](#).

- 29 Fields of an object type specify the representation of values of the object type by specifying the
30 component pieces from which it is composed (see [clause 7.4.1](#)). Static fields specify fields
31 associated with the object type itself (see [clause 7.4.3](#)). The fields of an object type are named
32 and they are typed via location signatures. The names of the members of the type are scoped to
33 the type (see [clause 7.5.2](#)). Fields are declared using a field definition (see [clause 7.11.2](#)).
- 34 Methods of an object type specify operations on values of the type (see [clause 7.4.2](#)). Static
35 methods specify operations on the type itself (see [clause 7.4.3](#)). Methods are named and they
36 have a method signature. The names of methods are scoped to the type (see [clause 7.5.2](#)).
37 Methods are declared using a method definition (see [clause 7.11.1](#)).
- 38 Properties of an object type specify named values that are accessible via methods that read and
39 write the value. The name of the property is the grouping of the methods; the methods themselves
40 are also named and typed via method signatures. The names of properties are scoped to the type
41 (see [clause 7.5.2](#)). Properties are declared using a property definition (see [clause 7.11.3](#)).
- 42 Events of an object type specify named state transitions in which subscribers may
43 register/unregister interest via accessor methods. When the state changes, the subscribers are
44 notified of the state transition. The name of the event is the grouping of the accessor methods;
45 the methods themselves are also named and typed via method signatures. The names of events
46 are scoped to the type (see [clause 7.5.2](#)). Events are declared using an event definition (see
47 [clause 7.11.4](#)).

1 7.9.6.4 Supporting Interface Contracts

2 Object type definitions may declare that they support zero or more interface contracts. Declaring support for an
3 interface contract places a requirement on the implementation of the object type to fully implement that
4 interface contract. Implementing an interface contract always reduces to implementing the required set of
5 methods, i.e. the methods required by the interface type.

6 The different types that the object type implements, i.e. the object type and any implemented interface types,
7 are each a separate logical grouping of named members. If a class `Foo` implements an interface `IFoo` and `IFoo`
8 declares a member method `int a()` and the class also declares a member method `int a()`, there are two
9 members, one in the `IFoo` interface type and one in the `Foo` class type. An implementation of `Foo` will provide
10 implementation for both, potentially shared.

11 Similarly, if a class implements two interfaces `IFoo` and `IBar` each of which defines a method `int a()` the
12 class will supply two method implementations, one for each interface, although they may share the actual code
13 of the implementation.

14 **CLS Rule 20:** CLS-compliant classes, value types, and interfaces shall not require the implementation of non-
15 CLS-compliant interfaces.

16 **Note:**

17 **CLS (consumer):** need not accept classes, value types or interfaces that violate this rule.

18 **CLS (extender):** need not provide syntax to author classes, value types, or interfaces that violate this rule.

19 **CLS (framework):** shall not externally expose classes, value types, or interfaces that violate this rule.

20 7.9.6.5 Supporting Class Contracts

21 Object type definitions may declare support for one other class contract. Declaring support for another class
22 contract is synonymous with object type inheritance (see [clause 7.9.9](#)).

23 7.9.6.6 Constructors

24 New values of an object type are created via **constructors**. Constructors shall be instance methods, defined via
25 a special form of method contract, which defines the method contract as a constructor for a particular object
26 type. The constructors for an object type are part of the object type definition. While the CTS and VES ensure
27 that only a properly defined constructor is used to make new values of an object type, the ultimate correctness
28 of a newly constructed object is dependent on the implementation of the constructor itself.

29 Object types shall define at least one constructor method, but that method need not be public. Creating a new
30 value of an object type by invoking a constructor involves the following steps in order:

- 31 1. Space for the new value is allocated in managed memory.
- 32 2. VES data structures of the new value are initialized and user-visible memory is zeroed.
- 33 3. The specified constructor for the object type is invoked.

34 Inside the constructor, the object type may do any initialization it chooses (possibly none).

35 **CLS Rule 21:** An object constructor shall call some class constructor of its base class before any access occurs
36 to inherited instance data. This does not apply to value types, which need not have constructors.

37 **CLS Rule 22:** An object constructor shall not be called except as part of the creation of an object, and an object
38 shall not be initialized twice.

39 **Note:**

40 **CLS (consumer):** Shall provide syntax for choosing the constructor to be called when an object is created.

41 **CLS (extender):** Shall provide syntax for defining constructor methods with different signatures. May issue a
42 compiler error if the constructor does not obey these rules.

CLS (framework): May assume that object creation includes a call to one of the constructors, and that no object is initialized twice. `System.MemberwiseClone` (see Partition IV) and deserialization (including object remoting) may not run constructors.

4 7.9.6.7 Finalizers

A class definition that creates an object type may supply an instance method to be called when an instance of the class is no longer accessible. The class `System.GC` (see [Partition IV](#)) provides limited control over the behavior of finalizers through the methods `SuppressFinalize` and `ReRegisterForFinalize`. Conforming implementations of the CLI may specify and provide additional mechanisms that affect the behavior of finalizers.

10 A conforming implementation of the CLI shall not automatically call a finalizer twice for the same object
11 unless

- there has been an intervening call to `ReRegisterForFinalize` (not followed by a call to `SuppressFinalize`), or
- the program has invoked an implementation-specific mechanism that is clearly specified to produce an alteration to this behavior

Rationale: Programmers expect that finalizers are run precisely once on any given object unless they take an explicit action to cause the finalizer to be run multiple times.

18 It is legal to define a finalizer for a Value Type. That finalizer however will only be run for *boxed* instances of
19 that Value Type.

Note: Since programmers may depend on finalizers to be called, the CLI should make every effort to ensure that finalizers are called, before it shuts down, for all objects that have not been exempted from finalization by a call to `SuppressFinalize`. The implementation should specify any conditions under which this behavior cannot be guaranteed.

Note: Since resources may become exhausted if finalizers are not called expeditiously, the CLI should ensure that finalizers are called soon after the instance becomes inaccessible. While relying on memory pressure to trigger finalization is acceptable, implementers should consider the use of additional metrics.

28 7.9.7 Value Type Definition

Not all types defined by a class definition are object types (see [clause 7.2.3](#)); in particular, value types are not object types but they are defined using a class definition. A class definition for a value type defines both the (unboxed) value type and the associated boxed type (see [clause 7.2.4](#)). The members of the class definition define the representation of both:

- When a non-static method (i.e. an instance or virtual method) is called on the value type its **this** pointer is a managed reference to the instance, whereas when the method is called on the associated boxed type the **this** pointer is an object reference.

Instance methods on value types receive a `this` pointer that is a managed pointer to the unboxed type whereas virtual methods (including those on interfaces implemented by the value type) receive an instance of the boxed type.

1. Value types do not support interface contracts, but their associated boxed types do.
2. A value type does not inherit; rather the base type specified in the class definition defines the base type of the boxed type.
3. The base type of a boxed type shall not have any fields.
4. Unlike object types, instances of value types do not require a constructor to be called when an instance is created. Instead, the verification rules require that verifiable code initialize instances to zero (null for object fields).

1 **7.9.8 Type Inheritance**

2 Inheritance of types is another way of saying that the derived type guarantees support for all of the type
3 contracts of the base type. In addition, the derived type usually provides additional functionality or specialized
4 behavior. A type inherits from a base type by implementing the type contract of the base type. An interface type
5 inherits from zero or more other interfaces. Value types do not inherit, although the associated boxed type is an
6 object type and hence inherits from other types

7 The derived class type shall support all of the supported interfaces contracts, class contracts, event contracts,
8 method contracts, and property contracts of its base type. In addition, all of the locations defined by the base
9 type are also defined in the derived type. The inheritance rules guarantee that code that was compiled to work
10 with a value of a base type will still work when passed a value of the derived type. Because of this, a derived
11 type also inherits the implementations of the base type. The derived type may extend, override, and/or hide
12 these implementations.

13 **7.9.9 Object Type Inheritance**

14 With the sole exception of `System.Object`, which does not inherit from any other object type, all object types
15 shall either explicitly or implicitly declare support for (inherit from) exactly one other object type. The graph of
16 the inherits-relation shall form a singly rooted tree with `System.Object` at the base, i.e. all object types
17 eventually inherit from the type `System.Object`.

18 An object type declares it shall not be used as a base type (be inherited from) by declaring that it is a **sealed**
19 type.

20 **CLS Rule 23:** `System.Object` is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-
21 compliant class.

22 Arrays are object types and as such inherit from other object types. Since arrays object types are manufactured
23 by the VES, the inheritance of arrays is fixed. See clause 7.9.1.

24 **7.9.10 Value Type Inheritance**

25 Value Types, in their unboxed form, do not inherit from any type. Boxed value types shall inherit directly from
26 `System.ValueType` unless they are enumerations, in which case they shall inherit from `System.Enum`. Boxed
27 value types shall be sealed.

28 Logically, the boxed type corresponding to a value type

- 29 • Is an object type.
- 30 • Will specify which object type is its base type, i.e. the object type from which it inherits.
- 31 • Will have a base type that has no fields defined.
- 32 • Will be sealed to avoid dealing with the complications of value slicing

33 The more restrictive rules specified here allow for more efficient implementation without severely
34 compromising functionality.

35 **7.9.11 Interface Type Inheritance**

36 Interface types may inherit from multiple interface types, i.e. an interface contract may list other interface
37 contracts that shall also be supported. Any type that implements support for an interface type shall also
38 implement support for all of the inherited interface types. This is different from object type inheritance in two
39 ways.

- 40 • Object types form a single inheritance tree; interface types do not.
- 41 • Object type inheritance specifies how implementations are inherited; interface type inheritance specifies
42 does not, since interfaces do not define implementation. Interface type inheritance specifies
43 additional contracts that an implementing object type shall support.

1 To highlight the last difference, consider an interface, `IFoo`, that has a single method. An interface, `IBar`, which
2 inherits from it is requiring that any object type that supports `IBar` also support `IFoo`. It does not say anything
3 about which methods `IBar` itself will have.

4 **7.10 Member Inheritance**

5 Only object types may inherit implementations, hence only object types may inherit members (see
6 clause 7.9.8). Interface types, while they do inherit from other interface types, only inherit the requirement to
7 implement method contracts, never fields or method implementations.

8 **7.10.1 Field Inheritance**

9 A derived object type inherits all of the non-static fields of its base object type. This allows instances of the
10 derived type to be used wherever instances of the base type are expected (the shapes, or layouts, of the
11 instances will be the same). Static fields are not inherited. Just because a field exists does not mean that it may
12 be read or written. The type visibility, field accessibility, and security attributes of the field definition (see
13 clause 7.5.3) determine if a field is accessible to the derived object type.

14 **7.10.2 Method Inheritance**

15 A derived object type inherits all of the instance and virtual methods of its base object type. It does not inherit
16 constructors or static methods. Just because a method exists does not mean that it may be invoked. It shall be
17 accessible via the typed reference that is being used by the referencing code. The type visibility, method
18 accessibility, and security attributes of the method definition (see clause 7.5.3) determine if a method is
19 accessible to the derived object type.

20 A derived object type may hide a non-virtual (i.e. static or instance) method of its base type by providing a new
21 method definition with the same name or same name and signature. Either method may still be invoked, subject
22 to method accessibility rules, since the type that contains the method always qualifies a method reference.

23 Virtual methods may be marked as `final`, in which case they shall not be overridden in a derived object type.
24 This ensures that the implementation of the method is available, by a virtual call, on any object that supports
25 the contract of the base class that supplied the final implementation. If a virtual method is not final it is possible
26 to demand a security permission in order to override the virtual method, so that the ability to provide an
27 implementation can be limited to classes that have particular permissions. When a derived type overrides a
28 virtual method, it may specify a new accessibility for the virtual method, but the accessibility in the derived
29 class shall permit at least as much access as the access granted to the method it is overriding. See clause 7.5.3.

30 **7.10.3 Property and Event Inheritance**

31 Properties and events are fundamentally constructs of the metadata intended for use by tools that target the CLI
32 and are not directly supported by the VES itself. It is, therefore, the job of the source language compiler and the
33 Reflection library [see Partition IV] to determine rules for name hiding, inheritance, and so forth. The source
34 compiler shall generate CIL that directly accesses the methods named by the events and properties, not the
35 events or properties themselves.

36 **7.10.4 Hiding, Overriding, and Layout**

37 There are two separate issues involved in inheritance. The first is which contracts a type shall implement and
38 hence which member names and signatures it shall provide. The second is the layout of the instance so that an
39 instance of a derived type can be substituted for an instance of any of its base types. Only the non-static fields
40 and the virtual methods that are part of the derived type affect the layout of an object.

41 The CTS provides independent control over both the names that are visible from a base type (**hiding**) and the
42 sharing of layout slots in the derived class (**overriding**). Hiding is controlled by marking a member in the
43 derived class as either `hide by name` or `hide by name-and-signature`. Hiding is always performed based on
44 the kind of member, that is, derived field names may hide base field names, but not method names, property
45 names, or event names. If a derived member is marked `hide by name`, then members of the same kind in the
46 base class with the same name are not visible in the derived class; if the member is marked `hide by name-and-`
47 `signature` then only a member of the same kind with exactly the same name and type (for fields) or method
48 signature (for methods) is hidden in the derived class. Implementation of the distinction between these two

1 forms of hiding is provided entirely by source language compilers and the Reflection library; it has no direct
2 impact on the VES itself.

3 For example:

```
4 class Base
5 { field int32      A;
6   field System.String A;
7   method int32      A();
8   method int32      A(int32);
9 }
10 class Derived inherits from Base
11 { field int32 A;
12   hidebysig method int32 A();
```

14 The member names available in type Derived are:

15 Table 3: Member names

Kind of member	Type / Signature of member	Name of member
Field	int32	A
Method	() -> int32	A
Method	(int32) -> int32	A

16 While hiding applies to all members of a type, overriding deals with object layout and is applicable only to
17 instance fields and virtual methods. The CTS provides two forms of member overriding, **new slot** and **expect**
18 **existing slot**. A member of a derived type that is marked as a new slot will always get a new slot in the object's
19 layout, guaranteeing that the base field or method is available in the object by using a qualified reference that
20 combines the name of the base type with the name of the member and its type or signature. A member of a
21 derived type that is marked as expect existing slot will re-use (i.e. share or override) a slot that corresponds to a
22 member of the same kind (field or method), name, and type if one already exists from the base type; if no such
23 slot exists, a new slot is allocated and used.

25 The general algorithm that is used for determining the names in a type and the layout of objects of the type is
26 roughly as follows:

- 27 • Flatten the inherited names (using the **hide by name** or **hide by name-and-signature** rule)
28 *ignoring* accessibility rules.
- 29 • For each new member that is marked "expect existing slot", look to see if an exact match on kind
30 (i.e. field or method), name, and signature exists and use that slot if it is found, otherwise allocate
31 a new slot.
- 32 • After doing this for all new members, add these new member-kind/name/signatures to the list of
33 members of this type
- 34 • Finally, remove any inherited names that match the new members based on the **hide by name** or
35 **hide by name-and-signature** rules.

36 7.11 Member Definitions

37 Object type definitions, interface type definitions, and value type definitions may include member definitions.
38 Field definitions define the representation of values of the type by specifying the substructure of the value.
39 Method definitions define operations on values of the type and operations on the type itself (static methods).
40 Property and event definitions may only be defined on object types. Property and events define named groups
41 of accessor method definitions that implement the named event or property behavior. Nested type declarations
42 define types whose names are scoped by the enclosing type and whose instances have full access to all
43 members of the enclosing class.

44 Depending on the kind of type definition, there are restrictions on the member definitions allowed.

1 7.11.1 Method Definitions

2 Method definitions are composed of a name, a method signature, and optionally an implementation of the
3 method. The method signature defines the calling convention, type of the parameters to the method, and the
4 return type of the method (see [clause 7.6.1](#)). The implementation is the code to execute when the method is
5 invoked. A value type or object type may define only one method of a given name and signature. However, a
6 derived object type may have methods that are of the same name and signature as its base object type. See
7 [clause 7.10.2](#) and [clause 7.10.4](#).

8 The name of the method is scoped to the type (see [clause 7.5.2](#)). Methods may be given accessibility attributes
9 (see [clause 7.5.3](#)). Methods may only be invoked with arguments that are assignment compatible with the
10 parameters types of the method signature. The return value of the method shall also be assignment compatible
11 with the location in which it is stored.

12 Methods may be marked as **static**, indicating that the method is not an operation on values of the type but
13 rather an operation associated with the type as a whole. Methods not marked as static define the valid
14 operations on a value of a type. When a non-static method is invoked, a particular value of the type, referred to
15 as **this** or the **this pointer**, is passed as an implicit parameter.

16 A method definition that does not include a method implementation shall be marked as **abstract**. All non-static
17 methods of an interface definition are abstract. Abstract method definitions are only allowed in object types that
18 are marked as abstract.

19 A non-static method definition in an object type may be marked as **virtual**, indicating that an alternate
20 implementation may be provided in derived types. All non-static method definitions in interface definitions
21 shall be virtual methods. Virtual method may be marked as **final**, indicating that derived object types are not
22 allowed to override the method implementation.

23 7.11.2 Field Definitions

24 Field definitions are composed of a name and a location signature. The location signature defines the type of
25 the field and the accessing constraints, see [clause 7.6.1](#). A value type or object type may define only one field
26 of a given name and type. However, a derived object type may have fields that are of the same name and type
27 as its base object type. See [clause 7.10.1](#) and [clause 7.10.4](#).

28 The name of the field is scoped to the type (see [clause 7.5.2](#)). Fields may be given accessibility attributes, see
29 [clause 7.5.3](#). Fields may only store values that are assignment compatible with the type of the field (see
30 [clause 7.3.1](#)).

31 Fields may be marked as **static**, indicating that the field is not part of values of the type but rather a location
32 associated with the type as a whole. Locations for the static fields are created when the type is loaded and
33 initialized when the type is initialized.

34 Fields not marked as static define the representation of a value of a type by defining the substructure of the
35 value (see [clause 7.4.1](#)). Locations for such fields are created within every value of the type whenever a new
36 value is constructed. They are initialized during construction of the new value. A non-static field of a given
37 name is always located at the same place within every value of the type.

38 A field that is marked **serializable** is to be serialized as part of the persistent state of a value of the type. This
39 standard does not specify the mechanism by which this is accomplished.

40 7.11.3 Property Definitions

41 A property definition defines a named value and the methods that access the value. A property definition
42 defines the accessing contracts on that value. Hence, the property definition specifies which accessing methods
43 exist and their respective method contracts. An implementation of a type that declares support for a property
44 contract shall implement the accessing methods required by the property contract. The implementation of the
45 accessing methods defines how the value is retrieved and stored.

46 A property definition is always part of either an interface definition or a class definition. The name and value of
47 a property definition is scoped to the object type or the interface type that includes the property definition.
48 While all of the attributes of a member may be applied to a property (accessibility, static, etc.) these are not
49 enforced by the CTS. Instead, the CTS requires that the method contracts that comprise the property shall

1 match the method implementations, as with any other method contract. There are no CIL instructions
2 associated with properties, just metadata.

3 By convention, properties define a **getter** method (for accessing the current value of the property) and
4 optionally a **setter** method (for modifying the current value of the property). The CTS places no restrictions on
5 the set of methods associated with a property, their names, or their usage.

6 **CLS Rule 24:** The methods that implement the **getter** and **setter** methods of a property shall be marked
7 **SpecialName** in the metadata.

8 **CLS Rule 25:** The accessibility of a property and of its accessors shall be identical.

9 **CLS Rule 26:** A property and its accessors shall all be static, all be virtual, or all be instance.

10 **CLS Rule 27:** The type of a property shall be the return type of the **getter** and the type of the last argument of
11 the **setter**. The types of the parameters of the property shall be the types of the parameters to the **getter** and
12 the types of all but the final parameter of the **setter**. All of these types shall be CLS-compliant, and shall not
13 be managed pointers (i.e. shall not be passed by reference).

14 **CLS Rule 28:** Properties shall adhere to a specific naming pattern. See [Section 9.4](#). The **SpecialName**
15 attribute referred to in CLS rule 26 shall be ignored in appropriate name comparisons and shall adhere to
16 identifier rules.

17 **Note:**

18 **CLS (consumer):** Shall ignore the **SpecialName** bit in appropriate name comparisons and shall adhere to
19 identifier rules. Otherwise, no direct support other than the usual access to the methods that define the
20 property.

21 **CLS (extender):** Shall ignore the **SpecialName** bit in appropriate name comparisons and shall adhere to
22 identifier rules. Otherwise, no direct support other than the usual access to the methods that define the
23 property. In particular, an extender need not be able to define properties.

24 **CLS (framework):** Shall design understanding that not all CLS languages will access the property using
25 special syntax.

26 7.11.4 Event Definitions

27 The CTS supports events in precisely the same way that it supports properties (see [clause 7.11.3](#)). The
28 conventional methods, however, are different and include means for subscribing and unsubscribing to events as
29 well as for firing the event.

30 **CLS Rule 29:** The methods that implement an event shall be marked **SpecialName** in the metadata.

31 **CLS Rule 30:** The accessibility of an event and of its accessors shall be identical.

32 **CLS Rule 31:** The **add** and **remove** methods for an event shall both either be present or absent.

33 **CLS Rule 32:** The **add** and **remove** methods for an event shall each take one parameter whose type defines the
34 type of the event and that shall be derived from **System.Delegate**.

35 **CLS Rule 33:** Events shall adhere to a specific naming pattern. See [Section 9.4](#). The **SpecialName** attribute
36 referred to in CLS rule 31 shall be ignored in appropriate name comparisons and shall adhere to identifier rules.

37 **Note:**

38 **CLS (consumer):** Shall ignore the **SpecialName** bit in appropriate name comparisons and shall adhere to
39 identifier rules. Otherwise, no direct support other than the usual access to the methods that define the event.

40 **CLS (extender):** Shall ignore the **SpecialName** bit in appropriate name comparisons and shall adhere to
41 identifier rules. Otherwise, no direct support other than the usual access to the methods that define the event.
42 In particular, an extender need not be able to define events.

43 **CLS (framework):** Shall design based on the understanding that not all CLS languages will access the event
44 using special syntax.

1 **7.11.5 Nested Type Definitions**

2 A nested type definition is identical to a top-level type definition, with one exception: a top-level type has a
3 visibility attribute, while the visibility of a nested type is the same as the visibility of the enclosing type. See
4 clause 7.5.3.

1 8 CLI Metadata

2 This section and its subsections contain only informative text, with the
3 exception of the CLS rules introduced here and repeated in Chapter 10. The
4 metadata format is specified in Partition II.

5 New types – value types and reference types – are introduced into the CTS via type declarations expressed in
6 metadata. In addition, metadata is a structured way to represent all information that the CLI uses to locate and
7 load classes, lay out instances in memory, resolve method invocations, translate CIL to native code, enforce
8 security, and set up runtime context boundaries. Every CLI PE/COFF module (see Partition II) carries a
9 compact metadata binary that is emitted into the module by the CLI-enabled development tool or compiler.

10 Each CLI-enabled language will expose a language-appropriate syntax for declaring types and members and for
11 annotating them with attributes that express which services they require of the infrastructure. Type imports are
12 also handled in a language-appropriate way, and it is the development tool or compiler that consumes the
13 metadata to expose the types that the developer sees.

14 Note that the typical component or application developer will not need to be aware of the rules for emitting and
15 consuming CLI metadata. While it may help a developer to understand the structure of metadata, the rules
16 outlined in this section are primarily of interest to tool builders and compiler writers.

17 8.1 Components and Assemblies

18 Each CLI component carries the metadata for declarations, implementations, and references specific to that
19 component. Therefore, the component-specific metadata is referred to as **component metadata**, and the
20 resulting component is said to be **self-describing**. In object models such as COM or CORBA, this information
21 is represented by a combination of typelibs, IDL files, DLLRegisterServer, and a myriad of custom files in
22 disparate formats and separate from the actual executable file. In contrast, the metadata is a fundamental part of
23 a CLI component.

24 Collections of CLI components and other files are packaged together for deployment into **assemblies**,
25 discussed in more detail in a later section. An assembly is a logical unit of functionality that serves as the
26 primary unit of reuse in the CLI. Assemblies establish a name scope for types.

27 Types declared and implemented in individual components are exported for use by other implementations via
28 the assembly in which the component participates. All references to a type are scoped by the identity of the
29 assembly in whose context the type is being used. The CLI provides services to locate a referenced assembly
30 and request resolution of the type reference. It is this mechanism that provides an isolation scope for
31 applications: the assembly alone controls its composition.

32 8.2 Accessing Metadata

33 Metadata is emitted into and read from a CLI module using either direct access to the file format as described
34 in Partition II or through the Reflection library. It is possible to create a tool that verifies a CLI module,
35 including the metadata, during development, based on the specifications supplied in Partition III and
36 Partition II.

37 When a class is loaded at runtime, the CLI loader imports the metadata into its own in-memory data structures,
38 which can be browsed via the CLI Reflection services. The Reflection services should be considered as similar
39 to a compiler; they automatically walk the inheritance hierarchy to obtain information about inherited methods
40 and fields, they have rules about hiding by name or name-and-signature, rules about inheritance of methods and
41 properties, and so forth.

42 8.2.1 Metadata Tokens

43 A metadata token is an implementation dependent encoding mechanism. Partition II describes the manner in
44 which metadata tokens are embedded in various sections of a CLI PE/COFF module. Metadata tokens are
45 embedded in CIL and native code to encode method invocations and field accesses at call sites; the token is

1 used by various infrastructure services to retrieve information from metadata about the reference and the type
2 on which it was scoped in order to resolve the reference.

3 A metadata token is a typed identifier of a metadata object (type declaration, member declaration, etc.). Given a
4 token, its type can be determined and it is possible to retrieve the specific metadata attributes for that metadata
5 object. However, a metadata token is not a persistent identifier. Rather it is scoped to a specific metadata
6 binary. A metadata token is represented as an index into a metadata data structure, so access is fast and direct.

7 8.2.2 Member Signatures in Metadata

8 Every location — including fields, parameters, method return values, and properties — has a type, and a
9 specification for its type is carried in metadata.

10 A value type describes values that are represented as a sequence of bits. A reference type describes values that
11 are represented as the location of a sequence of bits. The CLI provides an explicit set of built-in types, each of
12 which has a default runtime form as either a value type or a reference type. The metadata APIs may be used to
13 declare additional types, and part of the type specification of a variable encodes the identity of the type as well
14 as which form (value or reference) the type is to take at runtime.

15 Metadata tokens representing encoded types are passed to CIL instructions that accept a type (`newobj`,
16 `newarray`, `ldtoken`). See the CIL instruction set specification in [Partition III](#).

17 These encoded type metadata tokens are also embedded in member signatures. To optimize runtime binding of
18 field accesses and method invocations, the type and location signatures associated with fields and methods are
19 encoded into member signatures in metadata. A member signature embodies all of the contract information that
20 is used to decide whether a reference to a member succeeds or fails.

21 8.3 Unmanaged Code

22 It is possible to pass data from CLI managed code to unmanaged code. This always involves a transition from
23 managed to unmanaged code, which has some runtime cost, but data can often be transferred without copying.
24 When data must be reformatted the VES provides a reasonable specification of default behavior, but it is
25 possible to use metadata to explicitly require other forms of **marshalling** (i.e. reformatted copying). The
26 metadata also allows access to unmanaged methods through implementation-specific pre-existing mechanisms.

27 8.4 Method Implementation Metadata

28 For each method for which an implementation is supplied in the current CLI module, the tool or compiler will
29 emit information used by the CIL-to-native code compilers, the CLI loader, and other infrastructure services.
30 This information includes:

- 31 • Whether the code is managed or unmanaged.
- 32 • Whether the implementation is in native code or CIL (note that all CIL code is managed).
- 33 • The location of the method body in the current module, as an address relative to the start of the
34 module file in which it is located (a **Relative Virtual Address**, or **RVA**). Or, alternatively, the
35 RVA is encoded as 0 and other metadata is used to tell the infrastructure where the method
36 implementation will be found, including:
 - 37 o An implementation to be located via the CLI Interoperability Services. See related
38 specifications for details.
 - 39 o Forwarding calls through an imported global static method.

40 8.5 Class Layout

41 In the general case, the CLI loader is free to lay out the instances of a class in any way it chooses, consistent
42 with the rules of the CTS. However, there are times when a tool or compiler needs more control over the
43 layout. In the metadata, a class is marked with an attribute indicating whether its layout rule is:

- 1 • **autolayout**: A class marked “autolayout” indicates that the loader is free to lay out the class in
2 any way it sees fit; any layout information that may have been specified is ignored. This is the
3 default.
- 4 • **layoutsequential**: A class marked “layoutsequential” guides the loader to preserve field order as
5 emitted, but otherwise the specific offsets are calculated based on the CLI type of the field; these
6 may be shifted by explicit offset, padding, and/or alignment information.
- 7 • **explicitlayout**: A class marked “explicitlayout” causes the loader to ignore field sequence and to
8 use the explicit layout rules provided, in the form of field offsets and/or overall class size or
9 alignment. There are restrictions on legal layouts, specified in Partition II.

10 It is also possible to specify an overall size for a class. This enables a tool or compiler to emit a value type
11 specification where only the size of the type is supplied. This is useful in declaring CLI built-in types (such as
12 32 bit integer). It is also useful in situations where the data type of a member of a structured value type does not
13 have a representation in CLI metadata (e.g., C++ bit fields). In the latter case, as long as the tool or compiler
14 controls the layout, and CLI doesn't need to know the details or play a role in the layout, this is sufficient. Note
15 that this means that the VES can move bits around but can't marshal across machines – the emitting tool or
16 compiler will need to handle the marshaling.

17 Optionally, a developer may specify a packing size for a class. This is layout information that is not often used
18 but it allows a developer to control the alignment of the fields. It is not an alignment specification, per se, but
19 rather serves as a modifier that places a ceiling on all alignments. Typical values are 1, 2, 4, 8, or 16.

20 For the full specification of class layout attributes, see the classes in `System.Runtime.InteropServices` in
21 Partition IV.

22 8.6 **Assemblies: Name Scopes for Types**

23 An assembly is a collection of resources that are built to work together to deliver a cohesive set of
24 functionality. An assembly carries all of the rules necessary to ensure that cohesion. It is the unit of access to
25 resources in the CLI.

26 Externally, an assembly is a collection of exported resources, including types. Resources are exported by name.
27 Internally, an assembly is a collection of public (exported) and private (internal to the assembly) resources. It is
28 the assembly that determines which resources are to be exposed outside of the assembly and which resources
29 are accessible only within the current assembly scope. It is the assembly that controls how a reference to a
30 resource, public or private, is mapped onto the bits that implement the resource. For types in particular, the
31 assembly may also supply runtime configuration information. A CLI module can be thought of as a packaging
32 of type declarations and implementations, where the packaging decisions may change under the covers without
33 affecting clients of the assembly.

34 The identity of a type is its assembly scope and its declared name. A type defined identically in two different
35 assemblies is considered two different types.

36 **Assembly Dependencies**: An assembly may depend on other assemblies. This happens when implementations
37 in the scope of one assembly reference resources that are scoped in or owned by another assembly.

- 38 • All references to other assemblies are resolved under the control of the current assembly scope.
39 This gives an assembly an opportunity to control how a reference to another assembly is mapped
40 onto a particular version (or other characteristic) of that referenced assembly (although that target
41 assembly has sole control over how the referenced resource is resolved to an implementation).
- 42 • It is always possible to determine which assembly scope a particular implementation is running
43 in. All requests originating from that assembly scope are resolved relative to that scope.

44 From a deployment perspective, an assembly may be deployed by itself, with the assumption that any other
45 referenced assemblies will be available in the deployed environment. Or, it may be deployed with its dependent
46 assemblies.

47 **Manifests**: Every assembly has a manifest that declares what files make up the assembly, what types are
48 exported, and what other assemblies are required to resolve type references within the assembly. Just as CLI
49 components are self-describing via metadata in the CLI component, so are assemblies self-describing via their

1 manifests. When a single file makes up an assembly it contains both the metadata describing the types defined
2 in the assembly and the metadata describing the assembly itself. When an assembly contains more than one file
3 with metadata, each of the files describes the types defined in the file, if any, and one of these files also
4 contains the metadata describing the assembly (including the names of the other files, their cryptographic
5 hashes, and the types they export outside of the assembly).

6 **Applications:** Assemblies introduce isolation semantics for applications. An application is simply an assembly
7 that has an external entry point that triggers (or causes a hosting environment such as a browser to trigger) the
8 creation of a new Application Domain. This entry point is effectively the root of a tree of request invocations
9 and resolutions. Some applications are a single, self-contained assembly. Others require the availability of other
10 assemblies to provide needed resources. In either case, when a request is resolved to a module to load, the
11 module is loaded into the same Application Domain from which the request originated. It is possible to monitor
12 or stop an application via the Application Domain.

13 **References:** A reference to a type always qualifies a type name with the assembly scope within which the
14 reference is to be resolved – that is, an assembly establishes the name scope of available resources. However,
15 rather than establishing relationships between individual modules and referenced assemblies, every reference is
16 resolved through the current assembly. This allows each assembly to have absolute control over how references
17 are resolved. See Partition II.

18 8.7 Metadata Extensibility

19 CLI metadata is extensible. There are three reasons this is important:

20 • The Common Language Specification (CLS) is a specification for conventions that languages and
21 tools agree to support in a uniform way for better language integration. The CLS constrains parts
22 of the CTS model, and the CLS introduces higher-level abstractions that are layered over the
23 CTS. It is important that the metadata be able to capture these sorts of development-time
24 abstractions that are used by tools even though they are not recognized or supported explicitly by
25 the CLI.

26 • It should be possible to represent language-specific abstractions in metadata that are neither CIL
27 nor CLS language abstractions. For example, it should be possible, over time, to enable languages
28 like C++ to not require separate header files or IDL files in order to use types, methods, and data
29 members exported by compiled modules.

30 • It should be possible, in member signatures, to encode types and type modifiers that are used in
31 language-specific overloading. For example, to allow C++ to distinguish `int` from `long` even on
32 32-bit machines where both map to the underlying type `int32`.

33 This extensibility comes in the following forms:

34 • Every metadata object can carry custom attributes, and the metadata APIs provide a way to
35 declare, enumerate, and retrieve custom attributes. Custom attributes may be identified by a
36 simple name, where the value encoding is opaque and known only to the specific tool, language,
37 or service that defined it. Or, custom attributes may be identified by a type reference, where the
38 structure of the attribute is self-describing (via data members declared on the type) and any tool
39 including the CLI Reflection services may browse the value encoding.

40 **CLS Rule 34:** The CLS only allows a subset of the encodings of custom attributes. The only types that
41 shall appear in these encodings are (see Partition IV): `System.Type`, `System.String`, `System.Char`,
42 `System.Boolean`, `System.Byte`, `System.Int16`, `System.Int32`, `System.Int64`, `System.Single`,
43 `System.Double`, and any enumeration type based on a CLS-compliant base integer type.

44 **Note:**

45 **CLS (consumer):** Shall be able to read attributes encoded using the restricted scheme.

46 **CLS (extender):** Must meet all requirements for CLS consumer and be able to author new classes and
47 new attributes. Shall be able to attach attributes based on existing attribute classes to any metadata that
48 is emitted. Shall implement the rules for the `System.AttributeUsageAttribute` (see Partition IV).

1 **CLS (framework):** Shall externally expose only attributes that are encoded within the CLS rules and
2 following the conventions specified for `System.AttributeUsageAttribute`

3 • In addition to CTS type extensibility, it is possible to emit custom modifiers into member
4 signatures (see [Types in Partition II](#)). The CLI will honor these modifiers for purposes of method
5 overloading and hiding, as well as for binding, but will not enforce any of the language-specific
6 semantics. These modifiers can reference the return type or any parameter of a method, or the
7 type of a field. They come in two kinds: **required modifiers** that anyone using the member must
8 understand in order to correctly use it, and **optional modifiers** that may be ignored if the modifier
9 is not understood.

10 **CLS Rule 35:** The CLS does not allow publicly visible required modifiers (modreq, see [Partition II](#)), but
11 does allow optional modifiers (modopt, see [Partition II](#)) they do not understand.

12 **Note:**

13 **CLS (consumer):** Shall be able to read metadata containing optional modifiers and correctly copy
14 signatures that include them. May ignore these modifiers in type matching and overload resolution. May
15 ignore types that become ambiguous when the optional modifiers are ignored, or that use required
16 modifiers.

17 **CLS (extender):** Shall be able to author overrides for inherited methods with signatures that include
18 optional modifiers. Consequently, an extender must be able to copy such modifiers from metadata that it
19 imports. There is no requirement to support required modifiers, nor to author new methods that have any
20 kind of modifier in their signature.

21 **CLS (framework):** Shall not use required modifiers in externally visible signatures unless they are
22 marked as not CLS-compliant. Shall not expose two members on a class that differ only by the use of
23 optional modifiers in their signature unless only one is marked CLS-compliant.

24 8.8 **Globals, Imports, and Exports**

25 The CTS does not have the notion of **global statics**: all statics are associated with a particular class.
26 Nonetheless, the metadata is designed to support languages that rely on static data that is stored directly in a
27 PE/COFF file and accessed by its relative virtual address. In addition, while access to managed data and
28 managed functions is mediated entirely through the metadata itself, the metadata provides a mechanism for
29 accessing unmanaged data and unmanaged code.

30 **CLS Rule 36:** Global static fields and methods are not CLS-compliant.

31 **Note:**

32 **CLS (consumer):** Need not support global static fields or methods.

33 **CLS (extender):** Need not author global static fields or methods.

34 **CLS (framework):** Shall not define global static fields or methods.

35 8.9 **Scoped Statics**

36 The CTS does not include a model for file- or function-scoped static functions or data members. However,
37 there are times when a compiler needs a metadata token to emit into CIL for a scoped function or data member.
38 The metadata allows members to be marked so that they are never visible/accessible outside of the PE/COFF
39 file in which they are declared and for which the compiler guarantees to enforce all access rules.

40 **End informative text**

1 9 Name and Type Rules for the Common Language Specification

2 9.1 Identifiers

3 Languages that are either case-sensitive or case-insensitive can support the CLS. Since its rules apply only to
4 items exposed to other languages, private members or types that aren't exported from an assembly may use
5 any names they choose. For interoperation, however, there are some restrictions.

6 In order to make tools work well with a case-sensitive language it is important that the exact case of identifiers
7 be maintained. At the same time, when dealing with non-English languages encoded in Unicode, there may be
8 more than one way to represent precisely the same identifier that includes combining characters. The CLS
9 requires that identifiers obey the restrictions of the appropriate Unicode standard and persist them in Canonical
10 form C, which preserves case but forces combining characters into a standard representation. See CLS Rule 4,
11 in [Section 7.5.1](#).

12 At the same time, it is important that externally visible names not conflict with one another when used from a
13 case-insensitive programming language. As a result, all identifier comparisons shall be done internally to CLS-
14 compliant tools using the Canonical form KC, which first transforms characters to their case-canonical
15 representation. See CLS Rule 4, in [Section 7.5.1](#).

16 When a compiler for a CLS-compliant language supports interoperability with a non-CLS-compliant language
17 it must be aware that the CTS and VES perform all comparisons using code-point (i.e. byte-by-byte)
18 comparison. Thus, even though the CLS requires that persisted identifiers be in Canonical form C, references to
19 non-CLS identifiers will have to be persisted using whatever encoding the non-CLS language chose to use. It is
20 a language design issue, not covered by the CTS or the CLS, precisely how this should be handled.

21 9.2 Overloading

22 **Note:** The CTS, while it describes inheritance, object layout, name hiding, and overriding of virtual methods,
23 does not discuss overloading at all. While this is surprising, it arises from the fact that overloading is entirely
24 handled by compilers that target the CTS and not the type system itself. In the metadata, all references to types
25 and type members are fully resolved and include the precise signature that is intended. This choice was made
26 since every programming language has its own set of rules for coercing types and the VES does not provide a
27 means for expressing those rules.

28 Following the rules of the CTS, it is possible for duplicate names to be defined in the same scope as long as
29 they differ in either kind (field, method, etc.) or signature. The CLS imposes a stronger restriction for
30 overloading methods. Within a single scope, a given name may refer to any number of methods provided they
31 differ in any of the following:

- 32 • Number of parameters
- 33 • Type of each argument

34 Notice that the signature includes more information but CLS-compliant languages need not produce or
35 consume classes that differ only by that additional information (see [Partition II](#) for the complete list of
36 information carried in a signature):

- 37 • Calling convention
- 38 • Custom modifiers
- 39 • Return type
- 40 • Whether a parameter is passed by value or by reference (i.e. as a managed pointer or by-ref)

41 There is one exception to this rule. For the special names `op_Implicit` and `op_Explicit` described in
42 [clause 9.3.3](#) methods may be provided that differ only by their return type. These are marked specially and may
43 be ignored by compilers that don't support operator overloading.

Properties shall not be overloaded by type (that is, by the return type of their `getter` method), but they may be overloaded with different number or types of indices (that is, by the number and types of the parameters of its `getter` method). The overloading rules for properties are identical to the method overloading rules.

CLS Rule 37: Only properties and methods may be overloaded.

CLS Rule 38: Properties, instance methods, and virtual methods may be overloaded based only on the number and types of their parameters, except the conversion operators named `op_Implicit` and `op_Explicit` which may also be overloaded based on their return type.

Note:

CLS (consumer): May assume that only properties and methods are overloaded, and need not support overloading based on return type unless providing special syntax for operator overloading. If return type overloading isn't supported, then the `op_Implicit` and `op_Explicit` may be ignored since the functionality shall be provided in some other way by a CLS-compliant framework.

CLS (extender): Should not permit the authoring of overloads other than those specified here. It is not necessary to support operator overloading at all, hence it is possible to entirely avoid support for overloading on return type.

CLS (framework): Shall not publicly expose overloading except as specified here. Frameworks authors should bear in mind that many programming languages, including Object-Oriented languages, do not support overloading and will expose overloaded methods or properties through mangled names. Most languages support neither operator overloading nor overloading based on return type; so `op_Implicit` and `op_Explicit` shall always be augmented with some alternative way to gain the same functionality.

9.3 Operator Overloading

CLS-compliant consumer and extender tools are under no obligation to allow defining of operator overloading. CLS-compliant consumer and extender tools do not have to provide a special mechanism to call these methods.

Note: This topic is addressed by the CLS so that

- languages that do provide operator overloading can describe their rules in a way that other languages can understand, and
- languages that do not provide operator overloading can still access the underlying functionality without the addition of special syntax.

Operator overloading is described by using the names specified below, and by setting a special bit in the metadata (`SpecialName`) so that they do not collide with the user's name space. A CLS-compliant producer tool shall provide some means for setting this bit. If these names are used, they shall have precisely the semantics described here.

9.3.1 Unary Operators

Unary operators take one argument, perform some operation on it, and return the result. They are represented as static methods on the class that defines the type of their one operand or their return type. [Table 4: Unary Operator Names](#) shows the names that are defined.

Table 4: Unary Operator Names

Name	ISO C++ Operator Symbol
<code>op_Decrement</code>	Similar to <code>--</code>
<code>op_Increment</code>	Similar to <code>++</code>
<code>op_UnaryNegation</code>	<code>-</code> (unary)
<code>op_UnaryPlus</code>	<code>+</code> (unary)
<code>op_LogicalNot</code>	<code>!</code> ...
<code>op_True</code> ¹	Not defined

<code>op_False</code>	Not defined
<code>op_AddressOf</code>	<code>&</code> (unary)
<code>op_OnesComplement</code>	<code>~</code>
<code>op_PointerDereference</code>	<code>*</code> (unary)

¹ The `op_True` and `op_False` operators do not exist in C++. They are provided to support tri-state boolean types, such as those used in database languages.

³ 9.3.2 Binary Operators

⁴ Binary operators take two arguments, perform some operation and return a value. They are represented as static ⁵ methods on the class that defines the type of one of their two operands or the return type. ⁶ Table 5: Binary Operator Names shows the names that are defined.

⁷ Table 5: Binary Operator Names

Name	C++ Operator Symbol
<code>op_Addition</code>	<code>+</code> (binary)
<code>op_Subtraction</code>	<code>-</code> (binary)
<code>op_Multiply</code>	<code>*</code> (binary)
<code>op_Division</code>	<code>/</code>
<code>op_Modulus</code>	<code>%</code>
<code>op_ExclusiveOr</code>	<code>^</code>
<code>op_BitwiseAnd</code>	<code>&</code> (binary)
<code>op_BitwiseOr</code>	<code> </code>
<code>op_LogicalAnd</code>	<code>&&</code>
<code>op_LogicalOr</code>	<code> </code>
<code>op_Assign</code>	<code>=</code>
<code>op_LeftShift</code>	<code><<</code>
<code>op_RightShift</code>	<code>>></code>
<code>op_SignedRightShift</code>	Not defined
<code>op_UnsignedRightShift</code>	Not defined
<code>op_Equality</code>	<code>==</code>
<code>op_GreaterThan</code>	<code>></code>
<code>op_LessThan</code>	<code><</code>
<code>op_Inequality</code>	<code>!=</code>
<code>op_GreaterThanOrEqual</code>	<code>>=</code>
<code>op_LessThanOrEqual</code>	<code><=</code>
<code>op_UnsignedRightShiftAssignment</code>	Not defined
<code>op_MemberSelection</code>	<code>-></code>
<code>op_RightShiftAssignment</code>	<code>>>=</code>
<code>op_MultiplicationAssignment</code>	<code>*=</code>
<code>op_PointerToMemberSelection</code>	<code>->*</code>
<code>op_SubtractionAssignment</code>	<code>-=</code>
<code>op_ExclusiveOrAssignment</code>	<code>^=</code>
<code>op_LeftShiftAssignment</code>	<code><<=</code>

op_ModulusAssignment	%=
op_AdditionAssignment	+=
op_BitwiseAndAssignment	&=
op_BitwiseOrAssignment	=
op_Comma	,
op_DivisionAssignment	/=

9.3.3 Conversion Operators

Conversion operators are unary operations that allow conversion from one type to another. The operator method shall be defined as a static method on either the operand or return type. There are two types of conversions:

- An implicit (**widening**) coercion shall not lose any magnitude or precision. These should be provided using a method named `op_Implicit`
- An explicit (**narrowing**) coercion may lose magnitude or precision. These should be provided using a method named `op_Explicit`

Note: Conversions provide functionality that can't be generated in other ways, and many languages will not support the use of the conversion operators through special syntax. Therefore, CLS rules require that the same functionality be made available through an alternate mechanism. Using the more common `ToXXX` (where `XXX` is the target type) and `FromYYY` (where `YYY` is the name of the source type) naming pattern is recommended.

Because these operations may exist on the class of their operand type (so-called "from" conversions) and would therefore differ on their return type only, the CLS specifically allows that these two operators be overloaded based on their return type. The CLS, however, also requires that if this form of overloading is used then the language shall provide an alternate means for providing the same functionality since not all CLS languages will implement operators with special syntax.

CLS Rule 39: If either `op_Implicit` or `op_Explicit` is provided, an alternate means of providing the coercion shall be provided.

Note:

CLS (consumer): Where appropriate to the language design, use the existence of `op_Implicit` and/or `op_Explicit` in choosing method overloads and generating automatic coercions.

CLS (extender): Where appropriate to the language design, implement user-defined implicit or explicit coercion operators using the corresponding `op_Implicit`, `op_Explicit`, `ToXXX`, and/or `FromXXX` methods.

CLS (framework): If coercion operations are supported, they shall be provided as `FromXXX` and `ToXXX`, and optionally `op_Implicit` and `op_Explicit` as well. CLS frameworks are encouraged to provide such coercion operations.

9.4 Naming Patterns

See also [Partition V](#).

While the CTS does not dictate the naming of properties or events, the CLS does specify a pattern to be observed.

For Events:

An individual event is created by choosing or defining a delegate type that is used to signal the event. Then, three methods are created with names based on the name of the event and with a fixed signature. For the examples below we define an event named `Click` that uses a delegate type named `EventHandler`.

`EventAdd`, used to add a handler for an event

Pattern: `void add_<EventName> (<DelegateType> handler)`

Example: `void add_Click (EventHandler handler);`

```
1 EventRemove, used to remove a handler for an event
2     Pattern: void remove_<EventName> (<DelegateType> handler)
3     Example: void remove_Click (EventHandler handler);
4 EventRaise, used to signal that an event has occurred
5     Pattern: void family raise_<EventName> (Event e)
```

6 For Properties:

```
7 An individual property is created by deciding on the type returned by its getter method and the types of the
8 getter's parameters (if any). Then, two methods are created with names based on the name of the property and
9 these types. For the examples below we define two properties: Name takes no parameters and returns a
10 System.String, while Item takes a System.Object parameter and returns a System.Object. Item is referred
11 to as an indexed property, meaning that it takes parameters and thus may appear to the user as though it were
12 an array with indices
13 PropertyGet, used to read the value of the property
14     Pattern: <PropType> get_<PropName> (<Indices>)
15     Example: System.String get_Name ();
16     Example: System.Object get_Item (System.Object key);
17 PropertySet, used to modify the value of the property
18     Pattern: void set_<PropName> (<Indices>, <PropType>)
19     Example: void set_Name (System.String name);
20     Example: void set_Item (System.Object key, System.Object value);
```

21 9.5 Exceptions

22 The CLI supports an exception handling model, which is introduced in [clause 11.4.2](#). CLS compliant
23 frameworks may define and throw externally visible exceptions, but there are restrictions on the type of objects
24 thrown:

25 **CLS Rule 40:** Objects that are thrown shall be of type **System.Exception** or inherit from it. Nonetheless, CLS
26 compliant methods are not required to block the propagation of other types of exceptions.

27 **Note:**

28 **CLS (consumer):** Need not support throwing or catching of objects that are not of the specified type.

29 **CLS (extender):** Must support throwing of objects of type **System.Exception** or a type inheriting from it.
30 Need not support throwing of objects of other types.

31 **CLS (framework):** Shall not publicly expose thrown objects that are not of type **System.Exception** or a type
32 inheriting from it.

33 9.6 Custom Attributes

34 In order to allow languages to provide a consistent view of custom attributes across language boundaries, the
35 Base Class Library provides support for the following rules defined by the CLS:

36 **CLS Rule 41:** Attributes shall be of type **System.Attribute**, or inherit from it.

37 **Note:**

38 **CLS (consumer):** Need not support attributes that are not of the specified type.

39 **CLS (extender):** Must support the authoring of custom attributes.

40 **CLS (framework):** Shall not publicly expose attributes that are not of type **System.Attribute** or a type
41 inheriting from it.

1 The use of a particular attribute class may be restricted in various ways by placing an attribute on the attribute
2 class. The `System.AttributeUsageAttribute` is used to specify these restrictions. The restrictions supported
3 by the `System.AttributeUsageAttribute` are:

4 • What kinds of constructs (types, methods, assemblies, etc.) may have the attribute applied to
5 them. By default, instances of an attribute class can be applied to any construct. This is specified
6 by setting the value of the `ValidOn` property of `System.AttributeUsageAttribute`. Several
7 constructs may be combined.

8 • Multiple instances of the attribute class may be applied to a given piece of metadata. By default,
9 only one instance of any given attribute class can be applied to a single metadata item. The
10 `AllowMultiple` property of the attribute is used to specify the desired value.

11 • Do not inherit the attribute when applied to a type. By default, any attribute attached to a type
12 should be inherited to types that derive from it. If multiple instances of the attribute class are
13 allowed, the inheritance performs a union of the attributes inherited from the parent and those
14 explicitly applied to the child type. If multiple instance are not allowed, then an attribute of that
15 type applied directly to the child overrides the attribute supplied by the parent. This is specified
16 by setting the `Inherited` property of `System.AttributeUsageAttribute` to the desired value.

17 **Note:** Since these are CLS rules and not part of the CTS itself, tools are required to specify explicitly the
18 custom attributes they intend to apply to any given metadata item. That is, compilers or other tools that
19 generate metadata must implement the `AllowMultiple` and `Inherit` rules. The CLI does not supply attributes
20 automatically. The usage of attributes in the CLI is further described in [Partition II](#).

10 Collected CLS Rules

2 The complete set of CLS rules are collected here for reference. Recall that these rules apply only to "externally
3 visible" items – types that are visible outside of their own assembly and members of those types that have
4 public, family, or family-or-assembly accessibility. Furthermore, items may be explicitly marked as CLS-
5 compliant or not using the `System.CLSCompliantAttribute`. The CLS rules apply only to items that are
6 marked as CLS-compliant.

- 7 1. CLS rules apply only to those parts of a type that are accessible or visible outside of the defining
8 assembly (see [Section 6.3](#)).
- 9 2. Members of non-CLS compliant types shall not be marked CLS-compliant. (see [clause 6.3.1](#)).
- 10 3. The CLS does not include boxed value types (see [clause 7.2.4](#)).
- 11 4. Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 (ISBN 0-201-
12 61633-5) governing the set of characters permitted to start and be included in identifiers, available
13 on-line at <http://www.unicode.org/unicode/reports/tr15/tr15-18.html>. For CLS purposes, two
14 identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive,
15 1-1 lowercase mappings) are the same. That is, for two identifiers to be considered different under
16 the CLS they shall differ in more than simply their case. However, in order to override an inherited
17 definition the CLI requires the precise encoding of the original declaration be used (see
18 [clause 7.5.1](#)).
- 19 5. All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where
20 the names are identical and resolved via overloading. That is, while the CTS allows a single type
21 to use the same name for a method and a field, the CLS does not (see [clause 7.5.2](#)).
- 22 6. Fields and nested types shall be distinct by identifier comparison alone, even though the CTS
23 allows distinct signatures to be distinguished. Methods, properties, and events that have the same
24 name (by identifier comparison) shall differ by more than just the return type, except as specified in
25 CLS Rule 39 (see [clause 7.5.2](#)).
- 26 7. The underlying type of an enum shall be a built-in CLS integer type (see [clause 7.5.2](#)).
- 27 8. There are two distinct kinds of enums, indicated by the presence or absence of the
28 `System.FlagsAttribute` custom attribute. One represents named integer values, the other named
29 bit flags that can be combined to generate an unnamed value. The value of an enum is not limited
30 to the specified values (see [clause 7.5.2](#)).
- 31 9. Literal static fields of an enum shall have the type of the enum itself (see [clause 7.5.2](#)).
- 32 10. Accessibility shall not be changed when overriding inherited methods, except when overriding a
33 method inherited from a different assembly with accessibility Family-or-Assembly. In this case the
34 override shall have accessibility family (see [clause 7.5.3.2](#)).
- 35 11. All types appearing in a signature shall be CLS-compliant (see [clause 7.6.1](#)).
- 36 12. The visibility and accessibility of types and members shall be such that types in the signature of
37 any member shall be visible and accessible whenever the member itself is visible and accessible.
38 For example, a public method that is visible outside its assembly shall not have an argument whose
39 type is visible only within the assembly (see [clause 7.6.1](#)).
- 40 13. The value of a literal static is specified through the use of field initialization metadata (see
41 [Partition II](#)). A CLS compliant literal must have a value specified in field initialization metadata
42 that is of exactly the same type as the literal (or of the underlying type, if that literal is an `enum`).
43 (see [clause 7.6.1.2](#)).
- 44 14. Typed references are not CLS-compliant (see [clause 7.6.1.3](#)).
- 45 15. The varargs constraint is not part of the CLS, and the only calling convention supported by the CLS
46 is the standard managed calling convention (see [clause 7.6.1.5](#)).

- 1 16. Arrays shall have elements with a CLS-compliant type and all dimensions of the array shall have
2 lower bounds of zero. Only the fact that an item is an array and the element type of the array shall
3 be required to distinguish between overloads. When overloading is based on two or more array
4 types the element types shall be named types. (see clause 7.9.1).
- 5 17. Unmanaged pointer types are not CLS-compliant (see clause 7.9.2).
- 6 18. CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in order to
7 implement them (see clause 7.9.4).
- 8 19. CLS-compliant interfaces shall not define static methods, nor shall they define fields (see
9 clause 7.9.4).
- 10 20. CLS-compliant classes, value types, and interfaces shall not require the implementation of non-
11 CLS-compliant interfaces (see clause 7.9.6.4).
- 12 21. An object constructor shall call some class constructor of its base class before any access occurs to
13 inherited instance data. This does not apply to value types, which need not have constructors (see
14 clause 7.9.6.6).
- 15 22. An object constructor shall not be called except as part of the creation of an object, and an object
16 shall not be initialized twice (see clause 7.9.6.6).
- 17 23. `System.Object` is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-
18 compliant class (see clause 7.9.9).
- 19 24. The methods that implement the getter and setter methods of a property shall be marked
20 `SpecialName` in the metadata (see Partition II (see clause 7.11.3)).
- 21 25. The accessibility of a property and of its accessors shall be identical (see clause 7.11.3).
- 22 26. A property and its accessors shall all be static, all be virtual, or all be instance (see clause 7.11.3).
- 23 27. The type of a property shall be the return type of the getter and the type of the last argument of
24 the setter. The types of the parameters of the property shall be the types of the parameters to the
25 getter and the types of all but the final parameter of the setter. All of these types shall be CLS-
26 compliant, and shall not be managed pointers (i.e. shall not be passed by reference) (see
27 clause 7.11.3).
- 28 28. Properties shall adhere to a specific naming pattern. See Section 9.4. The `SpecialName` attribute
29 referred to in CLS rule 26 shall be ignored in appropriate name comparisons and shall adhere to
30 identifier rules (see clause 7.11.3).
- 31 29. The methods that implement an event shall be marked `SpecialName` in the metadata (see
32 Partition II (see clause 7.11.4).
- 33 30. The accessibility of an event and of its accessors shall be identical (see clause 7.11.4).
- 34 31. The `add` and `remove` methods for an event shall both either be present or absent (see clause 7.11.4).
- 35 32. The `add` and `remove` methods for an event shall each take one parameter whose type defines the
36 type of the event and that shall be derived from `System.Delegate` (see clause 7.11.4).
- 37 33. Events shall adhere to a specific naming pattern. See Section 9.4. The `SpecialName` attribute
38 referred to in CLS rule 31 shall be ignored in appropriate name comparisons and shall adhere to
39 identifier rules (see clause 7.11.4).
- 40 34. The CLS only allows a subset of the encodings of custom attributes. The only types that shall
41 appear in these encodings are: `System.Type`, `System.String`, `System.Char`, `System.Boolean`,
42 `System.Byte`, `System.Int16`, `System.Int32`, `System.Int64`, `System.Single`, `System.Double`,
43 and any enumeration type based on a CLS-compliant base integer type (see Section 8.7).
- 44 35. The CLS does not allow publicly visible required modifiers (modreq, see Partition II), but does
45 allow optional modifiers (modopt, see Partition II) they do not understand (see Section 8.7).
- 46 36. Global static fields and methods are not CLS-compliant (see Section 8.8).

- 1 37. Only properties and methods may be overloaded (see Section 9.2).
- 2 38. Properties, instance methods, and virtual methods may be overloaded based only on the number and
3 types of their parameters, except the conversion operators named `op_Implicit` and `op_Explicit`
4 which may also be overloaded based on their return type (see Section 9.2).
- 5 39. If either `op_Implicit` or `op_Explicit` is overloaded on its return type, an alternate means of
6 providing the coercion shall be provided (see clause 9.3.3).
- 7 40. Objects that are thrown shall be of type `System.Exception` or inherit from it (see Section 9.5).
8 Nonetheless, CLS compliant methods are not required to block the propagation of other types of
9 exceptions.
- 10 41. Attributes shall be of type `System.Attribute`, or inherit from it (see Section 9.6).

11 Virtual Execution System

2 The Virtual Execution System (VES) provides an environment for executing managed code. It provides direct
3 support for a set of built-in data types, defines a hypothetical machine with an associated machine model and
4 state, a set of control flow constructs, and an exception handling model. To a large extent, the purpose of the
5 VES is to provide the support required to execute the Common Intermediate Language instruction set (see
6 Partition III).

11.1 Supported Data Types

7 The CLI directly supports the data types shown in Table 6: Data Types Directly Supported by the CLI. That is,
8 these data types can be manipulated using the CIL instruction set (see Partition III).

10 **Table 6: Data Types Directly Supported by the CLI**

Data Type	Description
<code>int8</code>	8-bit 2's complement signed value
<code>unsigned int8</code>	8-bit unsigned binary value
<code>int16</code>	16-bit 2's complement signed value
<code>unsigned int16</code>	16-bit unsigned binary value
<code>int32</code>	32-bit 2's complement signed value
<code>unsigned int32</code>	32-bit unsigned binary value
<code>int64</code>	64-bit 2's complement signed value
<code>unsigned int64</code>	64-bit unsigned binary value
<code>float32</code>	32-bit IEC 60559:1989 floating point value
<code>float64</code>	64-bit IEC 60559:1989 floating point value
<code>native int</code>	native size 2's complement signed value
<code>native unsigned int</code>	native size unsigned binary value, also unmanaged pointer
<code>F</code>	native size floating point number (internal to VES, not user visible)
<code>O</code>	native size object reference to managed memory
<code>&</code>	native size managed pointer (may point into managed memory)

11 The CLI model uses an evaluation stack. Instructions that copy values from memory to the evaluation stack are
12 "loads"; instructions that copy values from the stack back to memory are "stores". The full set of data types in
13 Table 6: Data Types Directly Supported by the CLI can be represented in memory. However, the CLI supports
14 only a subset of these types in its operations upon values stored on its evaluation stack – `int32`, `int64`, `native int`.
15 In addition the CLI supports an internal data type to represent floating point values on the internal evaluation
16 stack. The size of the internal data type is implementation-dependent. For further information on the treatment
17 of floating-point values on the evaluation stack, see clause 11.1.3 and Partition III. Short numeric values (`int8`,
18 `int16`, `unsigned int8`, `unsigned int16`) are widened when loaded (memory-to-stack) and narrowed when stored
19 (stack-to-memory). This reflects a computer model that assumes, for numeric and object references, memory
20 cells are 1, 2, 4, or 8 bytes wide but stack locations are either 4 or 8 bytes wide. User-defined value types may
21 appear in memory locations or on the stack and have no size limitation; the only built-in operations on them are
22 those that compute their address and copy them between the stack and memory.

23 The only CIL instructions with special support for short numeric values (rather than support for simply the 4 or
24 8 byte integral values) are:

25

26 • Load and store instructions to/from memory: `ldelem`, `ldind`, `stind`, `stelem`

- 1 • Data conversion: **conv**, **conv.ovf**
- 2 • Array creation: **newarr**

3 The signed integer (int8, int16, int32, int64, and native int) and the respective unsigned integer (unsigned int8,
4 unsigned int16, unsigned int32, unsigned int64, and native unsigned int) types differ only in how the bits of the
5 integer are interpreted. For those operations where an unsigned integer is treated differently from a signed
6 integer (e.g. comparisons or arithmetic with overflow) there are separate instructions for treating an integer as
7 unsigned (e.g. **cgt.un** and **add.ovf.u**).

8 This instruction set design simplifies CIL-to-native code (eg. JIT) compilers and interpreters of CIL by
9 allowing them to internally track a smaller number of data types. See clause 11.3.2.1.

10 As described below, CIL instructions do not specify their operand types. Instead, the CLI keeps track of
11 operand types based on data flow and aided by a stack consistency requirement described below. For example,
12 the single **add** instruction will add two integers or two floats from the stack.

13 11.1.1 Native Size: native int, native unsigned int, O and &

14 The native-size, or generic, types (native int, native unsigned int, O, and &) are a mechanism in the CLI for
15 deferring the choice of a value's size. These data types exist as CIL types. But the CLI maps each to the native
16 size for a specific processor. (For example, data type **I** would map to **int32** on a Pentium processor, but to **int64**
17 on an IA64 processor). So, the choice of size is deferred until JIT compilation or runtime, when the CLI has
18 been initialized and the architecture is known. This implies that field and stack frame offsets are also not known
19 at compile time. For languages like Visual Basic, where field offsets are not computed early anyway, this is not
20 a hardship. In languages like C or C++, where sizes must be known when source code is compiled, a
21 conservative assumption that they occupy 8 bytes is sometimes acceptable (for example, when laying out
22 compile-time storage).

23 11.1.1.1 Unmanaged Pointers as Type Native Unsigned Int

24 *Rationale: For languages like C, when compiling all the way to native code, where the size of a pointer is
25 known at compile time and there are no managed objects, the fixed-size unsigned integer types (unsigned int32
26 or unsigned int64) may serve as pointers. However choosing pointer size at compile time has its
27 disadvantages. If pointers were chosen to be 32 bit quantities at compile time, the code would be restricted to
28 4 gigabytes of address space, even if it were run on a 64 bit machine. Moreover, a 64 bit CLI would need to
29 take special care so those pointers passed back to 32-bit code would always fit in 32 bits. If pointers were
30 chosen at compile time to be 64 bits, the code would run on a 32 bit machine, but pointers in every data
31 structure would be twice as large as necessary on that CLI.*

32 *For other languages, where the size of a data type need not be known at compile time, it is desirable to defer
33 the choice of pointer size from compile time to CLI initialization time. In that way, the same CIL code can
34 handle large address spaces for those applications that need them, while also being able to reap the size
35 benefit of 32 bit pointers for those applications that do not need a large address space.*

36 The native unsigned int type is used to represent unmanaged pointers with the VES. The metadata allows
37 unmanaged pointers to be represented in a strongly typed manner, but these types are translated into type native
38 unsigned int for use by the VES.

39 11.1.1.2 Managed Pointer Types: O and &

40 The **O** datatype represents an object reference that is managed by the CLI. As such, the number of specified
41 operations is severely limited. In particular, references shall only be used on operations that indicate that they
42 operate on reference types (e.g. **ceq** and **ldind.ref**), or on operations whose metadata indicates that references
43 are allowed (e.g. **call**, **ldsfld**, and **stfld**).

44 The **&** datatype (managed pointer) is similar to the **O** type, but points to the interior of an object. That is, a
45 managed pointer is allowed to point to a field within an object or an element within an array, rather than to
46 point to the 'start' of object or array.

47 Object references (**O**) and managed pointers (**&**) may be changed during garbage collection, since the data to
48 which they refer may be moved.

Note: In summary, object references, or **O** types, refer to the "outside" of an object, or to an object as-a-whole. But managed pointers, or **&** types, refer to the interior of an object. The **&** types are sometimes called "by-ref types" in source languages, since passing a field of an object by reference is represented in the VES by using an **&** type to represent the type of the parameter.

In order to allow managed pointers to be used more flexibly, they are also permitted to point to areas that aren't under the control of the CLI garbage collector, such as the evaluation stack, static variables, and unmanaged memory. This allows them to be used in many of the same ways that unmanaged pointers (U) are used. Verification restrictions guarantee that, if all code is verifiable, a managed pointer to a value on the evaluation stack doesn't outlast the life of the location to which it points.

11.1.1.3 Portability: Storing Pointers in Memory

Several instructions, including `call`, `cpblk`, `initblk`, `ldind.*`, and `stind.*`, expect an address on the top of the stack. If this address is derived from a pointer stored in memory, there is an important portability consideration.

1. Code that stores pointers in a native sized integer or pointer location (types `native int`, `Q`, `native unsigned int`, or `&`) is always fully portable.
2. Code that stores pointers in an 8 byte integer (type `int64` or `unsigned int64`) *can* be portable. But this requires that a `conv.ovf.u` instruction be used to convert the pointer from its memory format before its use as a pointer. This may cause a runtime exception if run on a 32-bit machine.
3. Code that uses any smaller integer type to store a pointer in memory (`int8`, `unsigned int8`, `int16`, `unsigned int16`, `int32`, `unsigned int32`) is *never* portable, even though the use of a `unsigned int32` or `int32` will work correctly on a 32-bit machine.

11.1.2 Handling of Short Integer Data Types

The CLI defines an evaluation stack that contains either 4-byte or 8-byte integers, but a memory model that encompasses in addition 1-byte and 2-byte integers. To be more precise, the following rules are part of the CLI model:

- Loading from 1-byte or 2-byte locations (arguments, locals, fields, statics, pointers) expands to 4-byte values. For locations with a known type (e.g. local variables) the type being accessed determines whether the load sign-extends (signed locations) or zero-extends (unsigned locations). For pointer dereference (`ldind.*`), the instruction itself identifies the type of the location (e.g. `ldind.u1` indicates an unsigned location, while `ldind.i1` indicates a signed location).
- Storing into a 1-byte or 2-byte location truncates to fit and will not generate an overflow error. Specific instructions (`conv.ovf.*`) can be used to test for overflow before storing.
- Calling a method assigns values from the evaluation stack to the arguments for the method, hence it truncates just as any other store would when the actual argument is larger than the formal argument.
- Returning from a method assigns a value to an invisible return variable, so it also truncates as a store would when the type of the value returned is larger than the return type of the method. Since the value of this return variable is then placed on the evaluation stack, it is then sign-extended or zero-extended as would any other load. Note that this truncation followed by extending is *not* identical to simply leaving the computed value unchanged.

It is the responsibility of any translator from CIL to native machine instructions to make sure that these rules are faithfully modeled through the native conventions of the target machine. The CLI does not specify, for example, whether truncation of short integer arguments occurs at the call site or in the target method.

11.1.3 Handling of Floating Point Datatypes

Floating-point calculations shall be handled as described in IEC 60559:1989. This standard describes encoding of floating point numbers, definitions of the basic operations and conversion, rounding control, and exception handling.

1 The standard defines special values, **NaN**, (not a number), **+infinity**, and **-infinity**. These values are returned
2 on overflow conditions. A general principle is that operations that have a value in the limit return an
3 appropriate infinity while those that have no limiting value return **NaN**, but see the standard for details.

4 **Note:** The following examples show the most commonly encountered cases.

5 $X \bmod 0 = \text{NaN}$
6 $0 * +\infty = 0 * -\infty = \text{NaN}$
7 $(X / 0) = +\infty$, if $X > 0$
8 NaN , if $X = 0$
9 $-\infty$, if $X < 0$
10 $\text{NaN} \text{ op } X = X \text{ op } \text{NaN} = \text{NaN}$ for all operations
11 $(+\infty) + (+\infty) = (+\infty)$
12 $X / (+\infty) = 0$
13 $X \bmod (-\infty) = -X$
14 $(+\infty) \cdot (+\infty) = \text{NaN}$

15 **Note:** This standard does not specify the behavior of arithmetic operations on denormalized floating point
16 numbers, nor does it specify when or whether such representations should be created. This is in keeping with
17 IEC 60559:1989. In addition, this standard does not specify how to access the exact bit pattern of NaNs that
18 are created, nor the behavior when converting a NaN between 32-bit and 64-bit representation. All of this
19 behavior is deliberately left implementation-specific.

20 For purposes of comparison, infinite values act like a number of the correct sign but with a very large
21 magnitude when compared with finite values. **NaN** is ‘unordered’ for comparisons (see **clt**, **clt.un**).

22 While the IEC 60559:1989 standard also allows for exceptions to be thrown under unusual conditions (such as
23 overflow and invalid operand), the CLI does not generate these exceptions. Instead, the CLI uses the **NaN**,
24 **+infinity**, and **-infinity** return values and provides the instruction **ckfinite** to allow users to generate an
25 exception if a result is **NaN**, **+infinity**, or **-infinity**.

26 The rounding mode defined in IEC 60559:1989 shall be set by the CLI to “round to the nearest number.” and
27 neither the CIL nor the class library provide a mechanism for modifying this setting. Conforming
28 implementations of the CLI need not be resilient to external interference with this setting. That is, they need not
29 restore the mode prior to performing floating-point operations, but rather may rely on it having been set as part
30 of their initialization.

31 For conversion to integers, the default operation supplied by the CIL is “truncate towards zero”. There are class
32 libraries supplied to allow floating-point numbers to be converted to integers using any of the other three
33 traditional operations (**round** to nearest integer, **floor** (truncate towards **-infinity**), **ceiling** (truncate towards
34 **+infinity**)).

35 Storage locations for floating point numbers (statics, array elements, and fields of classes) are of fixed size. The
36 supported storage sizes are **float32** and **float64**. Everywhere else (on the evaluation stack, as arguments, as
37 return types, and as local variables) floating point numbers are represented using an internal floating-point type.
38 In each such instance, the nominal type of the variable or expression is either **R4** or **R8**, but its value may be
39 represented internally with additional range and/or precision. The size of the internal floating-point
40 representation is implementation-dependent, may vary, and shall have precision at least as great as that of the
41 variable or expression being represented. An implicit widening conversion to the internal representation from
42 **float32** or **float64** is performed when those types are loaded from storage. The internal representation is
43 typically the native size for the hardware, or as required for efficient implementation of an operation. The
44 internal representation shall have the following characteristics:

- 45 • The internal representation shall have precision and range greater than or equal to the nominal
46 type.
- 47 • Conversions to and from the internal representation shall preserve value.

48 **Note:** This implies that an implicit widening conversion from **float32** (or **float64**) to the internal representation,
49 followed by an explicit conversion from the internal representation to **float32** (or **float64**), will result in a value
50 that is identical to the original **float32** (or **float64**) value.

1 **Rationale:** This design allows the CLI to choose a platform-specific high-performance representation for
2 floating point numbers until they are placed in storage locations. For example, it may be able to leave floating
3 point variables in hardware registers that provide more precision than a user has requested. At the same time,
4 CIL generators can force operations to respect language-specific rules for representations through the use of
5 conversion instructions.

6 When a floating-point value whose internal representation has greater range and/or precision than its nominal
7 type is put in a storage location it is automatically coerced to the type of the storage location. This may involve
8 a loss of precision or the creation of an out-of-range value (NaN, +infinity, or -infinity). However, the value
9 may be retained in the internal representation for future use, if it is reloaded from the storage location without
10 having been modified. It is the responsibility of the compiler to ensure that the retained value is still valid at
11 the time of a subsequent load, taking into account the effects of aliasing and other execution threads (see
12 memory model section). This freedom to carry extra precision is not permitted, however, following the
13 execution of an explicit conversion (conv.r4 or conv.r8), at which time the internal representation must be
14 exactly representable in the associated type.

15 **Note:** To detect values that cannot be converted to a particular storage type, a conversion instruction (conv.r4,
16 or conv.r8) may be used, followed by a check for a non-finite value using ckfinite. To detect underflow when
17 converting to a particular storage type, a comparison to zero is required before and after the conversion.

18 **Note:** The use of an internal representation that is wider than float32 or float64 may cause differences in
19 computational results when a developer makes seemingly unrelated modifications to their code, the result of
20 which may be that a value is spilled from the internal representation (e.g. in a register) to a location on the
21 stack.

1 11.1.4 CIL Instructions and Numeric Types

2 This clause contains only informative text

3 Most CIL instructions that deal with numbers take their operands from the evaluation stack (see
4 clause 11.3.2.1), and these inputs have an associated type that is known to the VES. As a result, a single
5 operation like `add` can have inputs of any numeric data type, although not all instructions can deal with all
6 combinations of operand types. Binary operations other than addition and subtraction require that both
7 operands be of the same type. Addition and subtraction allow an integer to be added to or subtracted from a
8 managed pointer (types `&` and `O`). Details are specified in Partition II.

9 Instructions fall into the following categories:

10 **Numeric:** These instructions deal with both integers and floating point numbers, and consider integers to be
11 signed. Simple arithmetic, conditional branch, and comparison instructions fit in this category.

12 **Integer:** These instructions deal only with integers. Bit operations and unsigned integer division/remainder fit
13 in this category.

14 **Floating point:** These instructions deal only with floating point numbers.

15 **Specific:** These instructions deal with integer and/or floating point numbers, but have variants that deal
16 specially with different sizes and unsigned integers. Integer operations with overflow detection, data conversion
17 instructions, and operations that transfer data between the evaluation stack and other parts of memory (see
18 clause 11.3.2) fit into this category.

19 **Unsigned/unordered:** There are special comparison and branch instructions that treat integers as unsigned and
20 consider unordered floating point numbers specially (as in “branch if greater than or unordered”):

21 **Load constant:** The load constant (`ldc.*`) instructions are used to load constants of type `int32`, `int64`, `float32` or
22 `float64`. Native size constants (type `native int`) shall be created by conversion from `int32` (conversion from `int64`
23 would not be portable) using `conv.i` or `conv.u`.

24 **Table 7: CIL Instructions by Numeric Category** shows the CIL instructions that deal with numeric values,
25 along with the category to which they belong. Instructions that end in “.*” indicate all variants of the
26 instruction (based on size of data and whether the data is treated as signed or unsigned).

27

Table 7: CIL Instructions by Numeric Category

<code>add</code>	Numeric
<code>add.ovf.*</code>	Specific
<code>and</code>	Integer
<code>breq[.s]</code>	Numeric
<code>bge[.s]</code>	Numeric
<code>bge.un[.s]</code>	Unsigned/unordered
<code>bgt[.s]</code>	Numeric
<code>bgt.un[.s]</code>	Unsigned/unordered
<code>ble[.s]</code>	Numeric
<code>ble.un[.s]</code>	Unsigned/unordered
<code>blt[.s]</code>	Numeric
<code>blt.un[.s]</code>	Unsigned/unordered
<code>bne.un[.s]</code>	Unsigned/unordered
<code>ceq</code>	Numeric

<code>div</code>	Numeric
<code>div.un</code>	Integer
<code>ldc.*</code>	Load constant
<code>ldelem.*</code>	Specific
<code>ldind.*</code>	Specific
<code>mul</code>	Numeric
<code>mul.ovf.*</code>	Specific
<code>neg</code>	Integer
<code>newarr.*</code>	Specific
<code>not</code>	Integer
<code>or</code>	Integer
<code>rem</code>	Numeric
<code>rem.un</code>	Integer
<code>shl</code>	Integer

cgt	Numeric
cgt.un	Unsigned/unordered
ckfinite	Floating point
clt	Numeric
clt.un	Unsigned/unordered
conv.*	Specific
conv.ovf.*	Specific

shr	Integer
shr.un	Specific
stelem.*	Specific
stind.*	Specific
sub	Numeric
sub.ovf.*	Specific
xor	Integer

1 2 End informative text

3 3 11.1.5 CIL Instructions and Pointer Types

4 4 This clause contains only informative text

5 5 **Rationale:** Some implementations of the CLI will require the ability to track pointers to objects and to collect
6 6 objects that are no longer reachable (thus providing memory management by "garbage collection"). This
7 7 process moves objects in order to reduce the working set and thus will modify all pointers to those objects as
8 8 they move. For this to work correctly, pointers to objects may only be used in certain ways. The **O** (object
9 9 reference) and & (managed pointer) datatypes are the formalization of these restrictions.

10 10 The use of object references is tightly restricted in the CIL. They are used almost exclusively with the "virtual
11 11 object system" instructions, which are specifically designed to deal with objects. In addition, a few of the base
12 12 instructions of the CIL handle object references. In particular, object references can be:

- 1 13 Loaded onto the evaluation stack to be passed as arguments to methods (**ldloc**, **ldarg**), and stored
2 14 from the stack to their home locations (**stloc**, **starg**)
- 2 15 Duplicated or popped off the evaluation stack (**dup**, **pop**)
- 3 16 Tested for equality with one another, but not other data types (**beq**, **beq.s**, **bne**, **bne.s**, **ceq**)
- 4 17 Loaded-from / stored-into unmanaged memory, in type unmanaged code only (**ldind.ref**,
5 18 **stind.ref**)
- 5 19 Created as a null reference (**ldnull**)
- 6 20 Returned as a value (**ret**)

21 21 Managed pointers have several additional base operations.

- 1 22 Addition and subtraction of integers, in units of *bytes*, returning a managed pointer (**add**,
2 23 **add.ovf.u**, **sub**, **sub.ovf.u**)
- 2 24 Subtraction of two managed pointers to elements of the same array, returning the number of *bytes*
3 25 between them (**sub**, **sub.ovf.u**)
- 3 26 Unsigned comparison and conditional branches based on two managed pointers (**bge.un**,
4 27 **bge.un.s**, **bgt.un**, **bgt.un.s**, **ble.un**, **ble.un.s**, **blt.un**, **blt.un.s**, **cgt.un**, **clt.un**)

28 28 Arithmetic operations upon managed pointers are intended *only* for use on pointers to elements of the same
29 29 array. Other uses of arithmetic on managed pointers is unspecified.

30 30 **Rationale:** Since the memory manager runs asynchronously with respect to programs and updates managed
31 31 pointers, both the distance between distinct objects and their relative position can change.

32 32 End informative text

1 11.1.6 Aggregate Data

2 This clause contains only informative text

3 The CLI supports *aggregate data*, that is, data items that have sub-components (arrays, structures, or object
4 instances) but are passed by copying the value. The sub-components can include references to managed
5 memory. Aggregate data is represented using a *value type*, which can be instantiated in two different ways:

- 6 • **Boxed**: as an Object, carrying full type information at runtime, and typically allocated on the heap
7 by the CLI memory manager.
- 8 • **Unboxed**: as a “value type instance” that does *not* carry type information at runtime and that is
9 never allocated directly on the heap. It can be part of a larger structure on the heap – a field of a
10 class, a field of a boxed value type, or an element of an array. Or it can be in the local variables
11 or incoming arguments array (see [clause 11.3.2](#)). Or it can be allocated as a static variable or
12 static member of a class or a static member of another value type.

13 Because value type instances, specified as method arguments, are copied on method call, they do not have
14 “identity” in the sense that Objects (boxed instances of classes) have.

15 11.1.6.1 Homes for Values

16 The **home** of a data value is where it is stored for possible reuse. The CLI directly supports the following home
17 locations:

- 18 • An incoming argument
- 19 • A local variable of a method
- 20 • An instance field of an object or value type
- 21 • A static field of a class, interface, or module
- 22 • An array element

23 For each home location, there is a means to compute (at runtime) the address of the home location and a means
24 to determine (at JIT compile time) the type of a home location. These are summarized in [Table 8: Address and](#)
25 [Type of Home Locations](#).

26 Table 8: Address and Type of Home Locations

Type of Home	Runtime Address Computation	JITtime Type Determination
Argument	ldarga for by-value arguments or ldarg for by-reference arguments	Method signature
Local Variable	ldloca for by-value locals or ldloc for by-reference locals	Locals signature in method header
Field	ldflda	Type of field in the class, interface, or module
Static	ldsflda	Type of field in the class, interface, or module
Array Element	ldelema for single-dimensional zero-based arrays or call the instance method Address	Element type of array

27 In addition to homes, built-in values can exist in two additional ways (i.e. without homes):

- 28 1. as constant values (typically embedded in the CIL instruction stream using ldc.* instructions)
- 29 2. as an intermediate value on the evaluation stack, when returned by a method or CIL instruction.

1 **11.1.6.2 Operations on Value Type Instances**

2 Value type instances can be created, passed as arguments, returned as values, and stored into and extracted
3 from locals, fields, and elements of arrays (i.e., copied). Like classes, value types may have both static and non-
4 static members (methods and fields). But, because they carry no type information at runtime, value type
5 instances are not substitutable for items of type Object; in this respect, they act like the built-in types int, long,
6 and so forth. There are two operations, box and unbox, that convert between value type instances and Objects.

7 **11.1.6.2.1 Initializing Instances of Value Types**

8 There are three options for initializing the home of a value type instance. You can zero it by loading the address
9 of the home (see Table 8: Address and Type of Home Locations) and using the **initobj** instruction (for local
10 variables this is also accomplished by setting the **zero initialize** bit in the method's header). You can call a
11 user-defined constructor by loading the address of the home (see Table 8: Address and Type of Home
12 Locations) and then calling the constructor directly. Or you can copy an existing instance into the home, as
13 described in clause 11.1.6.2.

14 **11.1.6.2.2 Loading and Storing Instances of Value Types**

15 There are two ways to load a value type onto the evaluation stack:

- 16 • Directly load the value from a home that has the appropriate type, using an **ldarg**, **ldloc**, **ldfld**, or
17 **ldsfld** instruction
- 18 • Compute the address of the value type, then use an **ldobj** instruction

19 Similarly, there are two ways to store a value type from the evaluation stack:

- 20 • Directly store the value into a home of the appropriate type, using a **starg**, **stloc**, **stfld**, or **stsfld**
21 instruction
- 22 • Compute the address of the value type, then use a **stobj** instruction

23 **11.1.6.2.3 Passing and Returning Value Types**

24 Value types are treated just as any other value would be treated:

- 25 • **To pass a value type by value**, simply load it onto the stack as you would any other argument:
26 use **ldloc**, **ldarg**, etc., or call a method that returns a value type. To access a value type parameter
27 that has been passed by value use the **ldarga** instruction to compute its address or the **ldarg**
28 instruction to load the value onto the evaluation stack.
- 29 • **To pass a value type by reference**, load the address of the value type as you normally would (see
30 Table 8: Address and Type of Home Locations). To access a value type parameter that has been
31 passed by reference use the **ldarg** instruction to load the address of the value type and then the
32 **ldobj** instruction to load the value type onto the evaluation stack.
- 33 • **To return a value type**, just load the value onto an otherwise empty evaluation stack and then
34 issue a **ret** instruction.

35 **11.1.6.2.4 Calling Methods**

36 Static methods on value types are handled no differently from static methods on an ordinary class: use a **call**
37 instruction with a metadata token specifying the value type as the class of the method. Non-static methods (i.e.,
38 instance and virtual methods) are supported on value types, but they are given special treatment. A non-static
39 method on a class (rather than a value type) expects a **this** pointer that is an instance of that class. This makes
40 sense for classes, since they have identity and the **this** pointer represents that identity. Value types, however,
41 have identity only when boxed. To address this issue, the **this** pointer on a non-static method of a value type is
42 a by-ref parameter of the value type rather than an ordinary by-value parameter.

43 A non-static method on a value type may be called in the following ways:

- 44 • Given an unboxed instance of a value type, the compiler will know the exact type of the object
45 statically. The **call** instruction can be used to invoke the function, passing as the first parameter
46 (the **this** pointer) the address of the instance. The metadata token used with the **call** instruction
47 shall specify the value type itself as the class of the method.

1 • Given a boxed instance of a value type, there are three cases to consider:
2 o Instance or virtual methods introduced on the value type itself: unbox the instance and call
3 the method directly using the value type as the class of the method.
4 o Virtual methods inherited from a parent class: use the **callvirt** instruction and specify the
5 method on the **System.Object**, **System.ValueType** or **System.Enum** class as appropriate.
6 o Virtual methods on interfaces implemented by the value type: use the **callvirt** instruction
7 and specify the method on the interface type.

8 **11.1.6.2.5 Boxing and Unboxing**

9 **Box** and **unbox** are conceptually equivalent to (and may be seen in higher-level languages as) casting between
10 a value type instance and **System.Object**. Because they change data representations, however, boxing and
11 unboxing are like the widening and narrowing of various sizes of integers (the **conv** and **conv.ovf** instructions)
12 rather than the casting of reference types (the **isinst** and **castclass** instructions). The **box** instruction is a
13 widening (always typesafe) operation that converts a value type instance to **System.Object** by making a copy
14 of the instance and embedding it in a newly allocated object. **Unbox** is a narrowing (runtime exception may be
15 generated) operation that converts a **System.Object** (whose runtime type is a value type) to a value type
16 instance. This is done by computing the address of the embedded value type instance without making a copy of
17 the instance.

18 **11.1.6.2.6 Castclass and IsInst on Value Types**

19 Casting to and from value type instances isn't permitted (the equivalent operations are **box** and **unbox**). When
20 boxed, however, it is possible to use the **isinst** instruction to see whether a value of type **System.Object** is the
21 boxed representation of a particular class.

22 **11.1.6.3 Opaque Classes**

23 Some languages provide multi-byte data structures whose contents are manipulated directly by address
24 arithmetic and indirection operations. To support this feature, the CLI allows value types to be created with a
25 specified size but no information about their data members. Instances of these "opaque classes" are handled in
26 precisely the same way as instances of any other class, but the **ldfld**, **stfld**, **ldflda**, **ldsfld**, and **stsfld** instructions
27 shall not be used to access their contents.

28 **End informative text**

11.2 Module Information

Partition II provides details of the CLI PE file format. The CLI relies on the following information about each method defined in a PE file:

- The *instructions* composing the method body, including all exception handlers.
- The *signature* of the method, which specifies the return type and the number, order, parameter passing convention, and built-in data type of each of the arguments. It also specifies the native calling convention (this does *not* affect the CIL virtual calling convention, just the native code).
- The *exception handling array*. This array holds information delineating the ranges over which exceptions are filtered and caught. See Partition II and clause 11.4.2.
- The size of evaluation stack that the method will require.
- The size of the locals array that the method will require.
- A “zero init flag” that indicates whether the local variables and memory pool should be initialized by the CLI (see also localalloc).
- Type of each local variable in the form of a signature of the local variable array (called the “locals signature”).

In addition, the file format is capable of indicating the degree of portability of the file. There are two kinds of restrictions that may be described:

- Restriction to a specific (32-bit or 64-bit) native size for integers.
- Restriction to a specific “endian-ness” (i.e. whether bytes are stored left-to-right or right-to-left within a machine word).

By stating which restrictions are placed on executing the code, the CLI class loader can prevent non-portable code from running on an architecture that it cannot support.

11.3 Machine State

One of the design goals of the CLI is to hide the details of a method call frame from the CIL code generator. This allows the CLI (and not the CIL code generator) to choose the most efficient calling convention and stack layout. To achieve this abstraction, the call frame is integrated into the CLI. The machine state definitions below reflect these design choices, where machine state consists primarily of global state and method state.

11.3.1 The Global State

The CLI manages multiple concurrent threads of control (not necessarily the same as the threads provided by a host operating system), multiple managed heaps, and a shared memory address space.

Note: A thread of control can be thought of, somewhat simplistically, as a singly-linked list of *method states*, where a new state is created and linked back to the current state by a *method call* instruction – the traditional model of a stack-based calling sequence. Notice that this model of the thread of control doesn’t correctly explain the operation of *tail*, *jmp*, or *throw* instructions.

Figure 2: Machine State Model illustrates the machine state model, which includes threads of control, method states, and multiple heaps in a shared address space. Method state, shown separately in Figure 3: Method State, is an abstraction of the stack frame. Arguments and local variables are part of the method state, but they can contain Object References that refer to data stored in any of the managed heaps. In general, arguments and local variables are only visible to the executing thread, while instance and static fields and array elements may be visible to multiple threads, and modification of such values is considered a side-effect.

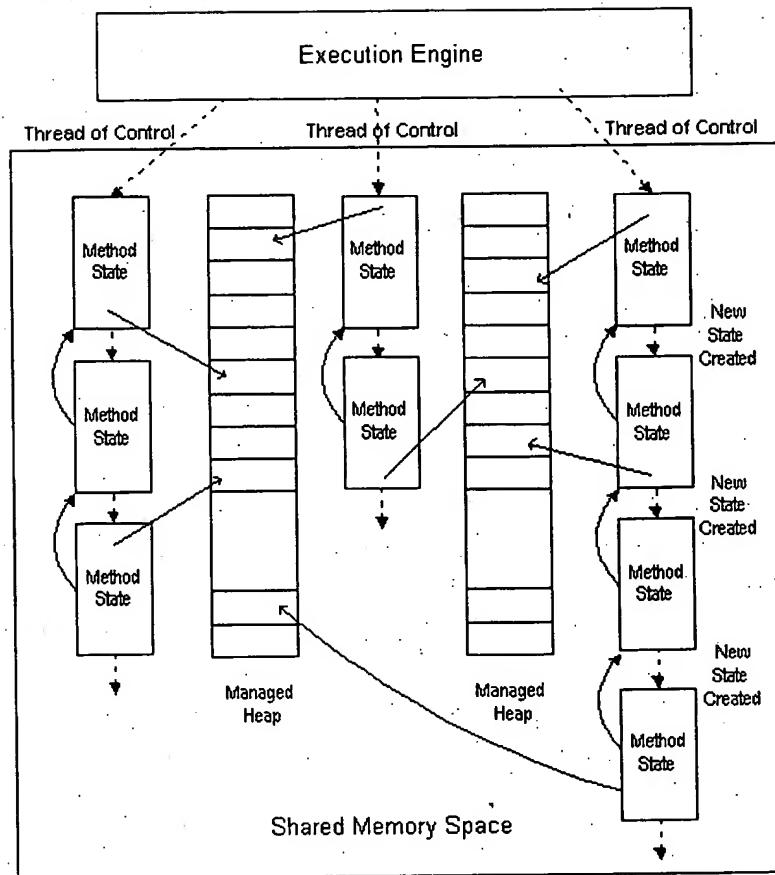


Figure 2: Machine State Model

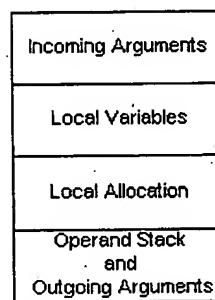


Figure 3: Method State

11.3.2 Method State

Method state describes the environment within which a method executes. (In conventional compiler terminology, it corresponds to a superset of the information captured in the "invocation stack frame"). The CLI method state consists of the following items:

- An instruction pointer (IP). This points to the next CIL instruction to be executed by the CLI in the present method.

- 1 • An *evaluation stack*. The stack is empty upon method entry. Its contents are entirely local to the
2 method and are preserved across call instructions (that's to say, if this method calls another, once
3 that other method returns, our evaluation stack contents are "still there"). The evaluation stack is
4 not addressable. At all times it is possible to deduce which one of a reduced set of types is stored
5 in any stack location at a specific point in the CIL instruction stream (see [clause 11.3.2.1](#)).
- 6 • A *local variable array* (starting at index 0). Values of local variables are preserved across calls
7 (in the same sense as for the evaluation stack). A local variable may hold any data type.
8 However, a particular slot shall be used in a type consistent way (where the type system is the one
9 described in [clause 11.3.2.1](#)). Local variables are initialized to 0 before entry if the initialize flag
10 for the method is set (see [Section 11.2](#)). The address of an individual local variable may be taken
11 using the `ldloca` instruction.
- 12 • An *argument array*. The values of the current method's incoming arguments (starting at index 0).
13 These can be read and written by logical index. The address of an argument can be taken using
14 the `ldarga` instruction. The address of an argument is also implicitly taken by the `arglist`
15 instruction for use in conjunction with typesafe iteration through variable-length argument lists.
- 16 • A *methodInfo* handle. This contains read-only information about the method. In particular it
17 holds the signature of the method, the types of its local variables, and data about its exception
18 handlers.
- 19 • A *local memory pool*. The CLI includes instructions for dynamic allocation of objects from the
20 local memory pool (`localalloc`). Memory allocated in the local memory pool is *addressable*. The
21 memory allocated in the local memory pool is reclaimed upon method context termination.
- 22 • A *return state* handle. This handle is used to restore the method state on return from the current
23 method. Typically, this would be the state of the method's caller. This corresponds to what in
24 conventional compiler terminology would be the *dynamic link*.
- 25 • A *security descriptor*. This descriptor is not directly accessible to managed code but is used by
26 the CLI security system to record security overrides (`assert`, `permit-only`, and `deny`).

27 The four areas of the method state – incoming arguments array, local variables array, local memory pool and
28 evaluation stack – are specified as if logically distinct areas. A conforming implementation of the CLI may map
29 these areas into one contiguous array of memory, held as a conventional stack frame on the underlying target
30 architecture, or use any other equivalent representation technique.

31 11.3.2.1 The Evaluation Stack

32 Associated with each method state is an evaluation stack. Most CLI instructions retrieve their arguments from
33 the evaluation stack and place their return values on the stack. Arguments to other methods and their return
34 values are also placed on the evaluation stack. When a procedure call is made the arguments to the called
35 methods become the incoming arguments array (see [clause 11.3.2.2](#)) to the method. This may require a memory
36 copy, or simply a sharing of these two areas by the two methods.

37 The evaluation stack is made up of slots that can hold any data type, including an unboxed instance of a value
38 type. The type state of the stack (the stack depth and types of each element on the stack) at any given point in a
39 program shall be identical for all possible control flow paths. For example, a program that loops an unknown
40 number of times and pushes a new element on the stack at each iteration would be prohibited.

41 While the CLI, in general, supports the full set of types described in [Section 11.1](#), the CLI treats the evaluation
42 stack in a special way. While some JIT compilers may track the types on the stack in more detail, the CLI only
43 requires that values be one of:

- 44 • `int64`, an 8-byte signed integer
- 45 • `int32`, a 4-byte signed integer
- 46 • `native int`, a signed integer of either 4 or 8 bytes, whichever is more convenient for the target
47 architecture

- F, a floating point value (float32, float64, or other representation supported by the underlying hardware)
- &, a managed pointer
- O, an object reference
- *, a “transient pointer,” which may be used only within the body of a single method, that points to a value known to be in unmanaged memory (see the CIL Instruction Set specification for more details. * types are generated internally within the CLI; they are not created by the user).
- A user-defined value type

The other types are synthesized through a combination of techniques:

- Shorter integer types in other memory locations are zero-extended or sign-extended when loaded onto the evaluation stack; these values are truncated when stored back to their home location.
- Special instructions perform numeric conversions, with or without overflow detection, between different sizes and between signed and unsigned integers.
- Special instructions treat an integer on the stack as though it were unsigned.
- Instructions that create pointers which are guaranteed not to point into the memory manager’s heaps (e.g. `ldloca`, `ldarga`, and `ldsflda`) produce transient pointers (type *) that may be used wherever a managed pointer (type &) or unmanaged pointer (type `native unsigned int`) is expected.
- When a method is called, an unmanaged pointer (type `native unsigned int` or *) is permitted to match a parameter that requires a managed pointer (type &). The reverse, however, is *not* permitted since it would allow a managed pointer to be “lost” by the memory manager.
- A managed pointer (type &) may be explicitly converted to an unmanaged pointer (type `native unsigned int`), although this is not verifiable and may produce a runtime exception.

11.3.2.2 Local Variables and Arguments

Part of each method state is an array that holds local variables and an array that holds arguments. Like the evaluation stack, each element of these arrays can hold any single data type or an instance of a value type. Both arrays start at 0 (that is, the first argument or local variable is numbered 0). The address of a local variable can be computed using the `ldloca` instruction, and the address of an argument using the `ldarga` instruction.

Associated with each method is metadata that specifies:

- whether the local variables and memory pool memory will be initialized when the method is entered
- the type of each argument and the length of the argument array (but see below for variable argument lists)
- the type of each local variable and the length of the local variable array.

The CLI inserts padding as appropriate for the target architecture. That is, on some 64-bit architectures all local variables may be 64-bit aligned, while on others they may be 8-, 16-, or 32-bit aligned. The CIL generator shall make no assumptions about the offsets of local variables within the array. In fact, the CLI is free to reorder the elements in the local variable array, and different JITters may choose to order them in different ways.

11.3.2.3 Variable Argument Lists

The CLI works in conjunction with the class library to implement methods that accept argument lists of unknown length and type (“varargs methods”). Access to these arguments is through a typesafe iterator in the Class Library, called `System.ArgIterator` (see [Partition IV](#)).

The CIL includes one instruction provided specifically to support the argument iterator, `arglist`. This instruction may be used only within a method that is declared to take a variable number of arguments. It returns

1 a value that is needed by the constructor for a `System.ArgIterator` object. Basically, the value created by
2 `arglist` provides access both to the address of the argument list that was passed to the method and a runtime
3 data structure that specifies the number and type of the arguments that were provided. This is sufficient for the
4 class library to implement the user visible iteration mechanism.

5 From the CLI point of view, varargs methods have an array of arguments like other methods. But only the
6 initial portion of the array has a fixed set of types and only these may be accessed directly using the `ldarg`,
7 `starg`, and `ldarga` instructions. The argument iterator allows access to both this initial segment and the
8 remaining entries in the array.

9 **11.3.2.4 Local Memory Pool**

10 Part of each method state is a local memory pool. Memory can be explicitly allocated from the local memory
11 pool using the `localalloc` instruction. All memory in the local memory pool is reclaimed on method exit, and that
12 is the only way local memory pool memory is reclaimed (there is no instruction provided to *free* local memory
13 that was allocated during this method invocation). The local memory pool is used to allocate objects whose
14 type or size is not known at compile time and which the programmer does not wish to allocate in the managed
15 heap.

16 Because the local memory pool cannot be shrunk during the lifetime of the method, a language implementation
17 cannot use the local memory pool for general-purpose memory allocation.

18 **11.4 Control Flow**

19 The CIL instruction set provides a rich set of instructions to alter the normal flow of control from one CIL
20 instruction to the next.

- 21 • **Conditional and Unconditional Branch** instructions for use within a method, provided the
22 transfer doesn't cross a protected region boundary (see [clause 11.4.2](#)).
- 23 • **Method call** instructions to compute new arguments, transfer them and control to a known or
24 computed destination method (see [clause 11.4.1](#)).
- 25 • **Tail call** prefix to indicate that a method should relinquish its stack frame before executing a
26 method call (see [clause 11.4.1](#)).
- 27 • **Return** from a method, returning a value if necessary.
- 28 • **Method jump** instructions to transfer the current method's arguments to a known or computed
29 destination method (see [clause 11.4.1](#)).
- 30 • **Exception-related instructions** (see [clause 11.4.2](#)). These include instructions to initiate an
31 exception, transfer control out of a protected region, and end a filter, catch clause, or finally
32 clause.

33 While the CLI supports control transfers within a method, there are several restrictions that shall be observed:

- 34 1. Control transfer is never permitted to enter a catch handler or finally clause (see [clause 11.4.2](#))
35 except through the exception handling mechanism.
- 36 2. Control transfer out of a protected region (see [clause 11.4.2](#)) is only permitted through an
37 exception instruction (`leave`, `end.filter`, `end.catch`, or `end.finally`).
- 38 3. The evaluation stack shall be empty after the return value is popped by a `ret` instruction.
- 39 4. Each slot on the stack shall have the same data type at any given point within the method body,
40 regardless of the control flow that allows execution to arrive there.
- 41 5. In order for the JIT compilers to efficiently track the data types stored on the stack, the stack shall
42 normally be empty at the instruction following an unconditional control transfer instruction (`br`,
43 `br.s`, `ret`, `jmp`, `throw`, `end.filter`, `end.catch`, or `end.finally`). The stack may be non-empty at
44 such an instruction only if at some earlier location within the method there has been a forward
45 branch to that instruction.

1 6. Control is not permitted to simply "fall through" the end of a method. All paths shall terminate
2 with one of these instructions: **ret**, **throw**, **jmp**, or (**tail**, followed by **call**, **calli**, or **callvirt**).

3 **11.4.1 Method Calls**

4 Instructions emitted by the CIL code generator contain sufficient information for different implementations of
5 the CLI to use different native calling convention. All method calls initialize the method state areas (see
6 clause 11.3.2) as follows:

7 1. The incoming arguments array is set by the caller to the desired values.
8 2. The local variables array always has **null** for Object types and for fields within value types that
9 hold objects. In addition, if the "zero init flag" is set in the method header, then the local
10 variables array is initialized to 0 for all integer types and 0.0 for all floating point types. Value
11 Types are not initialized by the CLI, but verified code will supply a call to an initializer as part of
12 the method's entry point code.
13 3. The evaluation stack is empty.

14 **11.4.1.1 Call Site Descriptors**

15 Call sites specify additional information that enables an interpreter or JIT compiler to synthesize any native
16 calling convention. All CIL calling instructions (**call**, **calli**, and **callvirt**) include a description of the call site.
17 This description can take one of two forms. The simpler form, used with the **calli** instruction, is a "call site
18 description" (represented as a metadata token for a stand-alone call signature) that provides:

19

- The number of arguments being passed.
- The data type of each argument.
- The order in which they have been placed on the call stack.
- The native calling convention to be used.

23 The more complicated form, used for the **call** and **callvirt** instructions, is a "method reference" (a metadata
24 **methodref** token) that augments the call site description with an identifier for the target of the call instruction.

25 **11.4.1.2 Calling Instructions**

26 The CIL has three call instructions that are used to transfer new argument values to a destination method.
27 Under normal circumstances, the called method will terminate and return control to the calling method.

28

- **call** is designed to be used when the destination address is fixed at the time the CIL is linked. In
29 this case, a method reference is placed directly in the instruction. This is comparable to a direct
30 call to a static function in C. It may be used to call static or instance methods or the (statically
31 known) superclass method within an instance method body.
- **calli** is designed for use when the destination address is calculated at run time. A method pointer
32 is passed on the stack and the instruction contains only the call site description.
- **callvirt**, part of the CIL common type system instruction set, uses the class of an object (known
34 only at runtime) to determine the method to be called. The instruction includes a method
35 reference, but the particular method isn't computed until the call actually occurs. This allows an
36 instance of a subclass to be supplied and the method appropriate for that subclass to be invoked.
37 The **callvirt** instruction is used both for instance methods and methods on interfaces. For further
38 details, see the Common Type System specification and the CIL Instruction Set specification.

40 In addition, each of these instructions may be immediately preceded by a **tail**. instruction prefix. This
41 specifies that the calling method terminates with this method call (and returns whatever value is returned by the
42 called method). The **tail**. prefix instructs the JIT compiler to discard the caller's method state prior to making
43 the call (if the call is from untrusted code to trusted code the frame cannot be fully discarded for security
44 reasons). When the called method executes a **ret** instruction, control returns not to the calling method but rather
45 to wherever that method would itself have returned (typically, return to caller's caller). Notice that the **tail**.

1 instruction shortens the lifetime of the caller's frame so it is unsafe to pass managed pointers (type &) as
2 arguments.

3 Finally, there are two instructions that indicate an optimization of the `tail`, case:

4 • `jmp` is followed by a `methodref` or `methoddef` token and indicates that the current method's state
5 should be discarded, its arguments should be transferred intact to the destination method, and
6 control should be transferred to the destination. The signature of the calling method shall exactly
7 match the signature of the destination method.

8 11.4.1.3 Computed Destinations

9 The destination of a method call may be either encoded directly in the CIL instruction stream (the `call` and `jmp`
10 instructions) or computed (the `callvirt`, and `calli` instructions). The destination address for a `callvirt` instruction
11 is automatically computed by the CLI based on the method token and the value of the first argument (the `this`
12 pointer). The method token shall refer to a virtual method on a class that is a direct ancestor of the class of the
13 first argument. The CLI computes the correct destination by locating the nearest ancestor of the first
14 argument's class that supplies an implementation of the desired method.

15 **Note:** The implementation can be assumed to be more efficient than the linear search implied here).

16 For the `calli` instruction the CIL code is responsible for computing a destination address and pushing it on the
17 stack. This is typically done through the use of a `ldftn` or `ldvirtfn` instruction at some earlier time. The `ldftn`
18 instruction includes a metadata token in the CIL stream that specifies a method, and the instruction pushes the
19 address of that method. The `ldvirtfn` instruction takes a metadata token for a virtual method in the CIL stream
20 and an object on the stack. It performs the same computation described above for the `callvirt` instruction but
21 pushes the resulting destination on the stack rather than calling the method.

22 The `calli` instruction includes a call site description that includes information about the native calling
23 convention that should be used to invoke the method. Correct CIL code shall specify a calling convention
24 specified in the `calli` instruction that matches the calling convention for the method that is being called.

25 11.4.1.4 Virtual Calling Convention

26 The CIL provides a "virtual calling convention" that is converted by the JIT into a native calling convention.
27 The JIT determines the optimal native calling convention for the target architecture. This allows the native
28 calling convention to differ from machine to machine, including details of register usage, local variable homes,
29 copying conventions for large call-by-value objects (as well as deciding, based on the target machine, what is
30 considered "large"). This also allows the JIT to reorder the values placed on the CIL virtual stack to match the
31 location and order of arguments passed in the native calling convention.

32 The CLI uses a single uniform calling convention for all method calls. It is the responsibility of the JITters to
33 convert this into the appropriate native calling convention. The contents of the stack at the time of a call
34 instruction (`call`, `calli`, or `callvirt` any of which may be preceded by `tail`.) are as follows:

35 1. If the method being called is an instance method (class or interface) or a virtual method, the `this`
36 pointer is the first object on the stack at the time of the call instruction. For methods on Objects
37 (including boxed value types), the `this` pointer is of type `O` (object reference). For methods on
38 value types, the `this` pointer is provided as a by-ref parameter; that is, the value is a pointer
39 (managed, &, or unmanaged, * or native int) to the instance.

40 2. The remaining arguments appear on the stack in left-to-right order (that is, the lexically leftmost
41 argument is the lowest on the stack, immediately following the `this` pointer, if any).
42 clause 11.4.1.5 describes how each of the three parameter passing conventions (by-value, by-
43 reference, and typed reference) should be implemented.

44 11.4.1.5 Parameter Passing

45 The CLI supports three kinds of parameter passing, all indicated in metadata as part of the signature of the
46 method. Each parameter to a method has its own passing convention (e.g., the first parameter may be passed
47 by-value while all others are passed by-ref). Parameters shall be passed in one of the following ways (see
48 detailed descriptions below):

- **By-value** parameters, where the **value** of an object is passed from the caller to the callee.
- **By-ref** parameters, where the **address** of the data is passed from the caller to the callee, and the type of the parameter is therefore a managed or unmanaged pointer.
- **Typed reference** parameters, where a runtime representation of the data type is passed along with the address of the data, and the type of the parameter is therefore one specially supplied for this purpose.

It is the responsibility of the CIL generator to follow these conventions. Verification checks that the types of parameters match the types of values passed, but is otherwise unaware of the details of the calling convention.

11.4.1.5.1 By-Value Parameters

For built-in types (integers, floats, etc.) the caller copies the value onto the stack before the call. For objects the object reference (type **O**) is pushed on the stack. For managed pointers (type **&**) or unmanaged pointers (type **native unsigned int**), the address is passed from the caller to the callee. For value types, see the protocol in [clause 11.1.6.2](#).

11.4.1.5.2 By-Ref Parameters

By-Ref Parameters are the equivalent of C++ reference parameters or PASCAL **var** parameters: instead of passing as an argument the value of a variable, field, or array element, its address is passed instead; and any assignment to the corresponding parameter actually modifies the corresponding caller's variable, field, or array element. Much of this work is done by the higher-level language, which hides from the user the need to compute addresses to pass a value and the use of indirection to reference or update values.

Passing a value by reference requires that the value have a home (see [clause 11.1.6.1](#)) and it is the address of this home that is passed. Constants, and intermediate values on the evaluation stack, cannot be passed as by-ref parameters because they have no home.

The CLI provides instructions to support by-ref parameters:

- calculate addresses of home locations (see [Table 8: Address and Type of Home Locations](#))
- load and store built-in data types through these address pointers (**Idind.***, **stind.***, **Idfld**, etc.)
- copy value types (**Idobj** and **cpobj**).

Some addresses (e.g., local variables and arguments) have lifetimes tied to that method invocation. These shall not be referenced outside their lifetimes, and so they should not be stored in locations that last beyond their lifetime. The CIL does not (and cannot) enforce this restriction, so the CIL generator shall enforce this restriction or the resulting CIL will not work correctly. For code to be verifiable (see [Section 7.8](#)) by-ref parameters may **only** be passed to other methods or referenced via the appropriate **stind** or **Idind** instructions.

11.4.1.5.3 Typed Reference Parameters

By-ref parameters and value types are sufficient to support statically typed languages (C++, Pascal, etc.). They also support dynamically typed languages that pay a performance penalty to box value types before passing them to polymorphic methods (Lisp, Scheme, SmallTalk, etc.). Unfortunately, they are not sufficient to support languages like Visual Basic that require by-reference passing of unboxed data to methods that are not statically restricted as to the type of data they accept. These languages require a way of passing *both* the address of the home of the data *and* the static type of the home. This is exactly the information that would be provided if the data were boxed, but without the heap allocation required of a box operation.

Typed reference parameters address this requirement. A typed reference parameter is very similar to a standard by-ref parameter but the static data type is passed as well as the address of the data. Like by-ref parameters, the argument corresponding to a typed reference parameter will have a home.

Note: If it were not for the fact that verification and the memory manager need to be aware of the data type and the corresponding address, a by-ref parameter could be implemented as a standard value type with two fields: the address of the data and the type of the data.

Like a regular by-ref parameter, a typed reference parameter can refer to a home that is on the stack, and that home will have a lifetime limited by the call stack. Thus, the CIL generator shall apply appropriate checks on

1 the lifetime of by-ref parameters; and verification imposes the same restrictions on the use of typed reference
2 parameters as it does on by-ref parameters (see [clause 11.4.1.5.2](#)).

3 A typed reference is passed by either creating a new typed reference (using the `mkrefany` instruction) or by
4 copying an existing typed reference. Given a typed reference argument, the address to which it refers can be
5 extracted using the `refanyval` instruction; the type to which it refers can be extracted using the `refanytype`
6 instruction.

7 11.4.1.5.4 Parameter Interactions

8 A given parameter may be passed using any one of the parameter passing conventions: by-value, by-ref, or
9 typed reference. No combination of these is allowed for a single parameter, although a method may have
10 different parameters with different calling mechanisms.

11 A parameter that has been passed in as typed reference shall not be passed on as by-ref or by-value without a
12 runtime type check and (in the case of by-value) a copy.

13 A by-ref parameter may be passed on as a typed reference by attaching the static type.

14 [Table 9: Parameter Passing Conventions](#) illustrates the parameter passing convention used for each data type.

15 **Table 9: Parameter Passing Conventions**

Type of data	Pass By	How data is sent
Built-in value type (int, float, etc.)	Value	Copied to called method, type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Typed reference	Address sent along with type information to called method
User-defined value type	Value	Called method receives a copy; type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Typed reference	Address sent along with type information to called method
Object	Value	Reference to data sent to called method, type statically known and class available from reference
	Reference	Address of reference sent to called method, type statically known and class available from reference
	Typed reference	Address of reference sent to called method along with static type information, class (i.e. dynamic type) available from reference

16 11.4.2 Exception Handling

17 Exception handling is supported in the CLI through exception objects and protected blocks of code. When an
18 exception occurs, an object is created to represent the exception. All exception objects are instances of some
19 class (i.e. they can be boxed value types, but not pointers, unboxed value types, etc.). Users may create their
20 own exception classes, typically by subclassing `System.Exception` (see [Partition IV](#)).

21 There are four kinds of handlers for protected blocks. A single protected block shall have exactly one handler
22 associated with it:

23

- 24 • A **finally handler** that shall be executed whenever the block exits, regardless of whether that
25 occurs by normal control flow or by an unhandled exception.
- 26 • A **fault handler** that shall be executed if an exception occurs, but not on completion of normal
27 control flow.
- 28 • A **type-filtered handler** that handles any exception of a specified class or any of its sub-classes.

1 • A user-filtered handler that runs a user-specified set of CIL instructions to determine whether
2 the exception should be ignored (i.e. execution should resume), handled by the associated handler,
3 or passed on to the next protected block.

4 Protected regions, the type of the associated handler, and the location of the associated handler and (if needed)
5 user-supplied filter code are described through an Exception Handler Table associated with each method. The
6 exact format of the Exception Handler Table is specified in detail in [Partition II](#). Details of the exception
7 handling mechanism are also specified in [Partition II](#).

8 11.4.2.1 Exceptions Thrown by the CLI

9 CLI instructions can throw the following exceptions as part of executing individual instructions. The
10 documentation for each instruction lists all the exceptions the instruction can throw (except for the general
11 purpose **ExecutionEngineException** described below that may be generated by all instructions).

12 Base Instructions (see [Partition III](#))

13 • **ArithmeticException**
14 • **DivideByZeroException**
15 • **ExecutionEngineException**
16 • **InvalidAddressException**
17 • **OverflowException**
18 • **SecurityException**
19 • **StackOverflowException**

20 Object Model Instructions (see [Partition III](#))

21 • **TypeLoadException**
22 • **IndexOutOfRangeException**
23 • **InvalidAddressException**
24 • **InvalidCastException**
25 • **MissingFieldException**
26 • **MissingMethodException**
27 • **NullReferenceException**
28 • **OutOfMemoryException**
29 • **SecurityException**
30 • **StackOverflowException**

31 The **ExecutionEngineException** is special. It can be thrown by any instruction and indicates an unexpected
32 inconsistency in the CLI. Running exclusively verified code can never cause this exception to be thrown by a
33 conforming implementation of the CLI. However, unverified code (even though that code is conforming CIL)
34 can cause this exception to be thrown if it corrupts memory. Any attempt to execute non-conforming CIL or
35 non-conforming file formats can cause completely unspecified behavior: a conforming implementation of the
36 CLI need not make any provision for these cases.

37 There are no exceptions for things like 'MetaDataTokenNotFound.' CIL verification (see [Partition V](#)) will
38 detect this inconsistency before the instruction is executed, leading to a verification violation. If the CIL is not
39 verified this type of inconsistency shall raise the generic **ExecutionEngineException**.

40 Exceptions can also be thrown by the CLI, as well as by user code, using the **throw** instruction. The handing of
41 an exception is identical, regardless of the source.

1 11.4.2.2 Subclassing Of Exceptions

2 Certain types of exceptions thrown by the CLI may be subclassed to provide more information to the user. The
3 specification of CIL instructions in [Partition III](#) describes what types of exceptions should be thrown by the
4 runtime environment when an abnormal situation occurs. Each of these descriptions allows a conforming
5 implementation to throw an object of the type described or an object of a subclass of that type.

6 **Note:** For instance, the specification of the `ckfinite` instruction requires that an exception of type
7 `ArithmeticException` or a subclass of `ArithmeticException` be thrown by the CLI. A conforming
8 implementation may simply throw an exception of type `ArithmeticException`, but it may also choose to
9 provide more information to the programmer by throwing an exception of type `NotFiniteNumberException`
10 with the offending number.

11 11.4.2.3 Resolution Exceptions

12 CIL allows types to reference, among other things, interfaces, classes, methods, and fields. Resolution errors
13 occur when references are not found or are mismatched. Resolution exceptions can be generated by references
14 from CIL instructions, references to base classes, to implemented interfaces, and by references from signatures
15 of fields, methods and other class members.

16 To allow scalability with respect to optimization, detection of resolution exceptions is given latitude such that it
17 may occur as early as install time and as late as execution time.

18 The latest opportunity to check for resolution exceptions from all references except CIL instructions is as part
19 of initialization of the type that is doing the referencing (see [Partition II](#)). If such a resolution exception is
20 detected the static initializer for that type, if present, shall not be executed.

21 The latest opportunity to check for resolution exceptions in CIL instructions is as part of the first execution of
22 the associated CIL instruction. When an implementation chooses to perform resolution exception checking in
23 CIL instructions as late as possible, these exceptions, if they occur, shall be thrown prior to any other non-
24 resolution exception that the VES may throw for that CIL instruction. Once a CIL instruction has passed the
25 point of throwing resolution errors (it has completed without exception, or has completed by throwing a non-
26 resolution exception), subsequent executions of that instruction shall no longer throw resolution exceptions.

27 If an implementation chooses to detect some resolution errors, from any references, earlier than the latest
28 opportunity for that kind of reference, it is not required to detect all resolution exceptions early.

29 An implementation that detects resolution errors early is allowed to prevent a class from being installed, loaded
30 or initialized as a result of resolution exceptions detected in the class itself or in the transitive closure of types
31 from following references of any kind.

32 For example, each of the following represents a permitted scenario. An installation program can throw
33 resolution exceptions (thus failing the installation) as a result of checking CIL instructions for resolution errors
34 in the set of items being installed. An implementation is allowed to fail to load a class as a result of checking
35 CIL instructions in a referenced class for resolution errors. An implementation is permitted to load and
36 initialize a class that has resolution errors in its CIL instructions.

37 The following exceptions are among those considered resolution exceptions:

- `BadImageFormatException`
- `EntryPointNotFoundException`
- `MissingFieldException`
- `MissingMemberException`
- `MissingMethodException`
- `NotSupportedException`
- `TypeLoadException`
- `TypeUnloadedException`

46 For example, when a referenced class cannot be found, a `TypeLoadException` is thrown. When a referenced
47 method (whose class is found) cannot be found, a `MissingMethodException` is thrown. If a matching method
48 being used consistently is accessible, but violates declared security policy, a `SecurityException` is thrown.

1 11.4.2.4 **Timing of Exceptions**

2 Certain types of exceptions thrown by CIL instructions may be detected before the instruction is executed. In
3 these cases, the specific time of the throw is not precisely defined, but the exception should be thrown no later
4 than the instruction is executed. That relaxation of the timing of exceptions is provided so that an
5 implementation may choose to detect and throw an exception before any code is run, e.g., at the time of CIL to
6 native code conversion.

7 There is a distinction between the time of detecting the error condition and throwing the associated exception.
8 An error condition may be detected early (e.g., at JIT time), but the condition may be signaled later (e.g. at the
9 execution time of the offending instruction) by throwing an exception.

10 The following exceptions are among those that may be thrown early by the runtime:

11 • MissingFieldException,
12 • MissingMethodException,
13 • SecurityException,
14 • TypeLoadException

15 11.4.2.5 **Overview of Exception Handling**

16 See the Exception Handling specification in [Partition II](#) for details.

17 Each method in an executable has associated with it a (possibly empty) array of exception handling
18 information. Each entry in the array describes a protected block, its filter, and its handler (which may be a
19 **catch** handler, a filter handler, a **finally** handler, or a **fault** handler). When an exception occurs, the CLI
20 searches the array for the first protected block that

21 • Protects a region including the current instruction pointer *and*
22 • Is a catch handler block *and*
23 • Whose filter wishes to handle the exception

24 If a match is not found in the current method, the calling method is searched, and so on. If no match is found
25 the CLI will dump a stack trace and abort the program.

26 **Note:** A debugger can intervene and treat this situation like a breakpoint, before performing any stack
27 unwinding, so that the stack is still available for inspection through the debugger.

28 If a match is found, the CLI walks the stack back to the point just located, but this time calling the **finally** and
29 **fault** handlers. It then starts the corresponding exception handler. Stack frames are discarded either as this
30 second walk occurs or after the handler completes, depending on information in the exception handler array
31 entry associated with the handling block.

32 Some things to notice are:

33 • The ordering of the exception clauses in the Exception Handler Table is important. If handlers
34 are nested, the most deeply nested try blocks shall come before the try blocks that enclose them.
35 • Exception handlers may access the local variables and the local memory pool of the routine that
36 catches the exception, but any intermediate results on the evaluation stack at the time the
37 exception was thrown are lost.
38 • An exception object describing the exception is automatically created by the CLI and pushed onto
39 the evaluation stack as the first item upon entry of a filter or catch clause.
40 • Execution cannot be resumed at the location of the exception, except with a **user-filtered**
41 **handler**.

42 11.4.2.6 **CIL Support for Exceptions**

43 The CIL has special instructions to:

44 • **Throw** and **rethrow** a user-defined exception.

- Leave a protected block and execute the appropriate **finally** clauses within a method, without throwing an exception. This is also used to exit a **catch** clause. Notice that leaving a protected block does **not** cause the fault clauses to be called.
- End a user-supplied filter clause (**endfilter**) and return a value indicating whether to handle the exception.
- End a finally clause (**endfinally**) and continue unwinding the stack.

11.4.2.7 Lexical Nesting of Protected Blocks

8 A protected region (also called a “try block”) is described by two addresses: the **trystart** is the address of the
9 first instruction to be protected and **tryend** is the address immediately following the last instruction to be
10 protected. A handler region is described by two addresses: the **handlerstart** is the address of the first
11 instruction of the handler and the **handlerend** is the address immediately following the last instruction of the
12 handler.

13 There are three kinds of handlers: catch, finally, and fault. A single exception entry consists of

- Optional: a type token (the type of exception to be handled) or **filterstart** (the address of the first instruction of the user-supplied filter code)
- Required: **protected region**
- Required: **handler region**.

18 Every method has associated with it a set of exception entries, called the **exception set**.

19 If an exception entry contains a **filterstart**, then **filterstart** < **handlerstart**. The **filter region** starts at the
20 instruction specified by **filterstart** and contains all instructions up to (but not including) that specified by
21 **handlerstart**. If there is no **filterstart** then the filter region is empty (hence does not overlap with any region).

22 No two regions (protected region, handler region, filter region) of a single exception entry may overlap with
23 one another.

24 For every pair of exception entries in an exception set, one of the following must be true:

- They **nest**: all three regions of one entry must be within a single region of the other entry.
- They are **disjoint**: all six regions of the two entries are pairwise disjoint (no addresses overlap)
- They **mutually protect**: the protected regions are the same and the other regions are pairwise disjoint.

29 The encoding of an exception entry in the file format (see Partition II) guarantees that only a catch handler (not
30 a fault handler or finally handler) can have a filter region.

11.4.2.8 Control Flow Restrictions on Protected Blocks

32 The following restrictions govern control flow into, out of, and between try blocks and their associated
33 handlers.

1. CIL code shall not enter a **filter**, **catch**, **fault** or **finally** block except through the CLI exception handling mechanism.
2. There are only two ways to enter a try block from outside its lexical body:
 - a. **Branching to or falling into the try block's first instruction.** The branch may be made using a conditional branch, an unconditional branch, or a **leave** instruction.
 - b. **Using a leave instruction from that try's catch block.** In this case, correct CIL code may branch to any instruction within the try block, not just its first instruction, so long as that branch target is not protected by yet another try, nested within the first
3. Upon entry to a try block the evaluation stack shall be empty.
4. The only ways CIL code may leave a try, filter, catch, finally or fault block are as follows:

- 1 a. **throw** from any of them.
- 2 b. **leave** from the body of a **try** or **catch** (in this case the destination of the **leave** shall have an empty evaluation stack and the **leave** instruction has the side-effect of emptying the evaluation stack).
- 3 c. **endfilter** may appear only as the lexically last instruction of a **filter** block, and it shall always be present (even if it is immediately preceded by a **throw** or other unconditional control flow). If reached, the evaluation stack shall contain an **int32** when the **endfilter** is executed, and the value is used to determine how exception handling should proceed.
- 4 d. **endfinally** from anywhere within a **finally** or **fault**, with the side-effect of emptying the evaluation stack.
- 5 e. **rethrow** from within a **catch** block, with the side-effect of emptying the evaluation stack.
- 6 5. When the **try** block is exited with a **leave** instruction, the evaluation stack shall be empty.
- 7 6. When a **catch** or **filter** clause is exited with a **leave** instruction, the evaluation stack shall be empty. This involves popping, from the evaluation stack, the exception object that was automatically pushed onto the stack.
- 8 7. CIL code shall not exit any **try**, **filter**, **catch** **finally** or **fault** block using a **ret** instruction.
- 9 8. The **localalloc** instruction cannot occur within an exception block: **filter**, **catch**, **finally**, or **fault**

18 11.5 Proxies and Remoting

19 A remoting boundary exists if it is not possible to share the identity of an object directly across the boundary. For example, if two objects exist on physically separate machines that do not share a common address space, then a remoting boundary will exist between them. There are other administrative mechanisms for creating remoting boundaries.

23 The VES provides a mechanism, called the **application domain**, to isolate applications running in the same operating system process from one another. Types loaded into one application domain are distinct from the same type loaded into another application domain, and instances of objects shall not be directly shared from one application domain to another. Hence, the application domain itself forms a remoting boundary.

27 The VES implements remoting boundaries based on the concept of a **proxy**. A proxy is an object that exists on one side of the boundary and represents an object on the other side. The proxy forwards references to instance fields and methods to the actual object for interpretation. Proxies do not forward references to static fields or calls to static methods.

31 The implementation of proxies is provided automatically for instances of types that derive from **System.MarshalByRefObject** (see [Partition IV](#)).

33 11.6 Memory Model and Optimizations

34 11.6.1 The Memory Store

35 By "memory store" we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a data object. The CLI accesses data objects in the memory store via the **ldind.*** and **stind.*** instructions.

38 11.6.2 Alignment

39 Built-in datatypes shall be *properly aligned*, which is defined as follows:

- 40 • 1-byte, 2-byte, and 4-byte data is properly aligned when it is stored at a 1-byte, 2-byte, or 4-byte boundary, respectively.
- 41 • 8-byte data is properly aligned when it is stored on the same boundary required by the underlying hardware for atomic access to a **native int**.

1 Thus, **int16** and **unsigned int16** start on even address; **int32**, **unsigned int32**, and **float32** start on an address
2 divisible by 4; and **int64**, **unsigned int64**, and **float64** start on an address divisible by 4 or 8, depending upon
3 the target architecture. The native size types (**native int**, **native unsigned int**, and **&**) are always naturally
4 aligned (4 bytes or 8 bytes, depending on architecture). When generated externally, these should also be aligned
5 to their natural size, although portable code may use 8 byte alignment to guarantee architecture independence.
6 It is strongly recommended that **float64** be aligned on an 8-byte boundary, even when the size of **native int** is
7 32 bits.

8 There is a special prefix instruction, **unaligned.**, that may immediately precede a **ldind**, **stind**, **initblk**, or **cpblk**
9 instruction. This prefix indicates that the data may have arbitrary alignment; the JIT is required to generate
10 code that correctly performs the effect of the instructions regardless of the actual alignment. Otherwise, if the
11 data is not properly aligned and no **unaligned.** prefix has been specified, executing the instruction may generate
12 unaligned memory faults or incorrect data.

13 11.6.3 Byte Ordering

14 For datatypes larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte
15 ordering may not run on all platforms. The PE file format (see [Section 11.2](#)) allows the file to be marked to
16 indicate that it depends on a particular type ordering.

17 11.6.4 Optimization

18 Conforming implementations of the CLI are free to execute programs using any technology that guarantees,
19 within a single thread of execution, that side-effects and exceptions generated by a thread are visible in the
20 order specified by the CIL. For this purpose volatile operations (including volatile reads) constitute side-
21 effects. Volatile operations are specified in [clause 11.6.7](#). There are no ordering guarantees relative to
22 exceptions injected into a thread by another thread (such exceptions are sometimes called “asynchronous
23 exceptions,” e.g., `System.Threading.ThreadAbortException`).

24 **Rationale:** *An optimizing compiler is free to reorder side-effects and synchronous exceptions to the extent that
25 this reordering does not change any observable program behavior.*

27 **Note:** An implementation of the CLI is permitted to use an optimizing compiler, for example, to convert CIL to
28 native machine code provided the compiler maintains (within each single thread of execution) the same order
29 of side-effects and synchronous exceptions.

30 This is a stronger condition than ISO C++ (which permits reordering between a pair of sequence points) or ISO
31 Scheme (which permits reordering of arguments to functions).

32 11.6.5 Locks and Threads

33 The logical abstraction of a thread of control is captured by an instance of the `System.Threading.Thread`
34 object in the class library. Classes beginning with the string “`System.Threading`” (see [Partition IV](#)) provide
35 much of the user visible support for this abstraction.

36 To create consistency across threads of execution, the CLI provides the following mechanisms:

- 37 1. **Synchronized methods.** A lock that is visible across threads controls entry to the body of a
38 synchronized method. For instance and virtual methods the lock is associated with the `this` pointer.
39 For static methods the lock is associated with the type to which the method belongs. The lock is
40 taken by the logical thread (see `System.Threading.Thread` in [Partition IV](#)) and may be entered
41 any number of times by the same thread; entry by other threads is prohibited while the first thread
42 is still holding the lock. The CLI shall release the lock when control exits (by any means) the
43 method invocation that first acquired the lock.
- 44 2. **Explicit locks and monitors.** These are provided in the class library, see
45 `System.Threading.Monitor`. Many of the methods in the `System.Threading.Monitor` class
46 accept an `object` as argument, allowing direct access to the same lock that is used by synchronized
47 methods. While the CLI is responsible for ensuring correct protocol when this lock is only used by
48 synchronized methods, the user must accept this responsibility when using explicit monitors on
49 these same objects.

- 1 3. **Volatile reads and writes.** The CIL includes a prefix, `volatile.`, that specifies that the
2 subsequent operation is to be performed with the cross-thread visibility constraints described in
3 clause 11.6.7. In addition, the class library provides methods to perform explicit volatile reads
4 (name) and writes (name), as well as a barrier synchronization (name)
- 5 4. **Built-in atomic reads and writes.** All reads and writes of certain properly aligned data types are
6 guaranteed to occur atomically. See clause 11.6.6.
- 7 5. **Explicit atomic operations.** The class library provides a variety of atomic operations in the
8 System.Threading.Interlocked class.

9 Acquiring a lock (`System.Threading.Monitor.Enter` or entering a synchronized method) shall implicitly
10 perform a volatile read operation, and releasing a lock (`System.Threading.Monitor.Exit` or leaving a
11 synchronized method) shall implicitly perform a volatile write operation. See clause 11.6.7.

12 11.6.6. Atomic Reads and Writes

13 A conforming CLI shall guarantee that read and write access to *properly aligned* memory locations no larger
14 than the native word size (the size of type `native int`) is atomic (see clause 11.6.2). Atomic writes shall alter no
15 bits other than those written. Unless explicit layout control (see Partition II (Controlling Instance Layout)) is
16 used to alter the default behavior, data elements no larger than the natural word size (the size of a `native int`)
17 shall be properly aligned. Object references shall be treated as though they are stored in the native word size.

18 **Note:** There is no guarantee about atomic update (read-modify-write) of memory, except for methods provided
19 for that purpose as part of the class library (see Partition IV). An atomic write of a “small data item” (an item
20 no larger than the native word size) is required to do an atomic read/write/modify on hardware that does not
21 support direct writes to small data items.

22 **Note:** There is no guaranteed atomic access to 8-byte data when the size of a `native int` is 32 bits even though
23 some implementations may perform atomic operations when the data is aligned on an 8-byte boundary.

25 11.6.7. Volatile Reads and Writes

26 The `volatile.` prefix on certain instructions shall guarantee cross-thread memory ordering rules. They do not
27 provide atomicity, other than that guaranteed by the specification of clause 11.6.6.

28 A volatile read has “acquire semantics” meaning that the read is guaranteed to occur prior to any references to
29 memory that occur after the read instruction in the CIL instruction sequence. A volatile write has “release
30 semantics” meaning that the write is guaranteed to happen after any memory references prior to the write
31 instruction in the CIL instruction sequence.

32 A conforming implementation of the CLI shall guarantee this semantics of volatile operations. This ensures
33 that all threads will observe volatile writes performed by any other thread in the order they were performed. But
34 a conforming implementation is *not* required to provide a single total ordering of volatile writes as seen from
35 all threads of execution.

36 An optimizing compiler that converts CIL to native code shall not remove any volatile operation, nor may it
37 coalesce multiple volatile operations into a single operation.

38 **Rationale:** One traditional use of volatile operations is to model hardware registers that are visible through
39 direct memory access. In these cases, removing or coalescing the operations may change the behavior of the
40 program.

42 **Note:** An optimizing compiler from CIL to native code is permitted to reorder code, provided that it guarantees
43 both the single-thread semantics described in Section 11.6 and the cross-thread semantics of volatile operations.

44 11.6.8. Other Memory Model Issues

45 All memory allocated for static variables (other than those assigned RVAs within a PE file, see Partition II) and
46 objects shall be zeroed before they are made visible to any user code.

1 A conforming implementation of the CLI shall ensure that, even in a multi-threaded environment and without
2 proper user synchronization, objects are allocated in a manner that prevents unauthorized memory access and
3 prevents illegal operations from occurring. In particular, on multiprocessor memory systems where explicit
4 synchronization is required to ensure that all relevant data structures are visible (for example, vtable pointers)
5 the EE shall be responsible for either enforcing this synchronization automatically or for converting errors due
6 to lack of synchronization into non-fatal, non-corrupting, user-visible exceptions.

7 It is explicitly *not* a requirement that a conforming implementation of the CLI guarantee that all state updates
8 performed within a constructor be uniformly visible before the constructor completes. CIL generators may
9 ensure this requirement themselves by inserting appropriate calls to the memory barrier or volatile write
10 instructions.

11.7 Atomicity of Memory Accesses

12 The CLI makes several assumptions about atomicity of memory references, and these translate directly into
13 rules required of either programmers or translators from high-level languages into CIL.

- 14 • Read and write access to word-length memory locations (types **native int** and **native unsigned**
15 **int**) that are properly aligned is atomic. Correct translation from CIL to native code requires
16 generation of native code sequences that supply this atomicity guarantee. There is no guarantee
17 about atomic update (read-modify-write) of memory, except for methods provided for that
18 purpose as part of the class library (see Partition IV).
- 19 • Read and write access to 4-byte data (**int32** and **unsigned int32**) that is aligned on a 4-byte
20 boundary is atomic, even on a 64-bit machine. Again, there is no guarantee about atomic read-
21 modify-write.
- 22 • One- and Two-byte data that does not cross a word boundary will be read atomically, but writing
23 may write the entire word back to memory.
- 24 • No other memory references are performed atomically.

25 When the CLI controls the layout of managed data, it pads the data so that if an object starts at a word boundary
26 all of the fields that require 4 or fewer bytes will be aligned so that reads will be atomic. The managed heap
27 always aligns data that it allocates to maintain this rule, so heap references (type **O**) to data that does not have
28 explicit layout will occur atomically where possible. Similarly, static variables of managed classes are allocated
29 so that they, too, are aligned when possible. The CLI aligns stack frames to word boundaries, but need not
30 attempt to align to an 8-byte boundary on 32-bit machines even if the frame contains 8-byte values.

12 Index

2 2 & 16
3 abstract 50, 51, 57
4 abstract class 32
5 access 39
6 accessible 39
7 application 63
8 application domain 63, 97
9 array element 9, 22, 36
10 array type 9, 36, 46
11 assembly 9, 38, 40, 51, 60, 62
12 Assembly 16
13 assembly dependency 62
14 assembly scope 9, 38
15 assignment compatibility 42
16 assignment compatible 35, 38
17 *Attributes* 9
18 autolayout 62
19 behavior 32
20 box 34
21 boxed type 10, 34
22 boxed value 10, 34
23 byref 43
24 by-ref 76
25 Calling convention 10
26 cast 10, 36
27 explicit 68
28 implicit 68
29 narrowing 68
30 widening 68
31 casting 10, 36
32 class contract 10, 41
33 class definitions 49
34 class type 10, 41, 49
35 CLS 11, 24, 27
36 CLS-compliant 29
37 coercion 35
38 COFF *See PE*
39 Common Intermediate Language *See CIL*
40 Common Language Specification *See CLS*
41 component metadata 60
42 compound type 11, 36
43 concrete 11, 51
44 constructor 52
45 contract 12, 30, 41, 61
46 conversions
47 explicit 36
48 implicit 36
49 Custom attributes 12
50 enclosing type 40, 41
51 enum 38
52 enumeration type 38
53 Enumerations 12
54 equal 34
55 equality 34, 35
56 event contract 42
57 exact type 33, 37
58 expect existing slot 56
59 explicitlayout 62
60 exportable 40
61 family 9, 40
62 family-and-assembly 9, 40
63 family-or-assembly 9, 40
64 field 9, 22, 36
65 final 55, 57
66 fully describe 33
67 garbage collection 26
68 getter 13, 58
69 global statics 64
70 hide 55, 56, 57, 64
71 by name 55
72 by name and signature 55
73 id 14

1 identical	34	39 method contract	42
2 Identifiers	14	40 method signature	44
3 identity	34, 35, 36	41 names	
4 implicit type	45	42 special	66
5 indexed property	<i>See</i> property, indexed	43 narrowing	35, 68
6 inherit	32, 41	44 nested	40
7 inheritance demand	41	45 nested type	40, 41
8 inherits	36	46 new slot	56
9 init-only	43	47 null	36, 37
10 init-only constraint	43	48 object	33
11 instance field	37	49 object type	32, 33, 51
12 Instance methods	14	50 OOP	<i>See</i> Programming, Object-Oriented
13 interface contract	41	51 optional modifiers	64
14 interface definition	48	52 overriding	37, 55
15 interface type	32, 33, 41	53 parameter signature	44
16 Intermediate Language	<i>See</i> CIL	54 partial description	32
17 kind	38	55 PE	60
18 Label	20	56 pointer type	22, 32, 48
19 Code label	11	57 Pointers	
20 layout	55	58 Managed	16
21 layoutsequential	62	59 private	9, 40
22 literal	39, 43	60 programming	
23 literal constraint	43	61 object-oriented	30
24 local signature	43	62 property	
25 location	35	63 indexed	69
26 location constraint	43	64 property contract	42
27 location signature	43	65 proxy	97
28 managed code	25	66 public	9, 40, 51
29 managed data	16, 25, 26	67 publicly accessible parts	29
30 managed pointer	43	68 qualified name	19, 38
31 manifest	62	69 qualify	38
32 marshalling	61	70 rank	19, 46
33 member	36	71 reference	63
34 member scope	21, 38	72 reference demand	41
35 member signatures	61	73 reference type	32
36 messages	32	74 referenced entity	39
37 metadata	60	75 referent	39
38 method	33	76 Relative Virtual Address	<i>See</i> RVA

1	remoting boundary.....	20, 97	28	typedref.....	See typed reference
2	representation	32	29	typeless programming.....	32
3	required modifiers.....	64	30	typesafe.....	21, 45
4	RVA.....	61	31	unbox	34
5	scope.....	20, 38, 51	32	uniqueness	38
6	sealed.....	20, 54	33	name	38
7	self-describing	60	34	validation	
8	serializable.....	20, 57	35	metadata.....	60
9	setter	20, 58	36	Validation	22
10	signature	12, 20, 41, 42	37	value type.....	32
11	static.....	57	38	Value types	22
12	static field	21, 37	39	vararg	22
13	static method.....	21, 37	40	varargs	44
14	Super call	21	41	vector	47
15	System.AttributeUsageAttribute.....	70	42	verification	25, 45
16	System.Enum.....	54	43	Verification	23
17	System.TypedReference	See typed reference	44	VES	See Virtual Execution System
18	System.ValueType.....	54	45	virtual	37, 57
19	this	57	46	virtual call	23, 37
20	this pointer.....	21, 36, 37, 57	47	Virtual Execution System	24
21	type	32	48	virtual method	37
22	type definer.....	21, 45	49	Virtual methods	23
23	type definition.....	21, 38	50	visibility	51
24	type name.....	21, 38	51	visible	39
25	type safety.....	25, 45	52	volatile constraint	43
26	type signature.....	22, 42	53	widening	35, 68
27	typed reference	44			
54					
55					

THIS PAGE BLANK (USPTO)